PolyORB User's Guide

Version 2.7.0w-20100926 (rev. 165103)
2010-09-03

Robert Duff, Jérôme Hugues, Laurent Pautet,
Thomas Quinot, Samuel Tardieu

Table of Contents

About This Guide

This guide describes the use of PolyORB, a middleware that enables the construction of distributed Ada applications.

It describes the features of the middleware and related APIs and tools, and details how to use them to build Ada applications.

What This Guide Contains

This guide contains the following chapters:

- Chapter 1 [Introduction to PolyORB], page 3 provides a brief description of middleware and PolyORB's architecture.
- Chapter 2 [Installation], page 7 details how to configure and install PolyORB on your system.
- Chapter 3 [Overview of PolyORB personalities], page 11 enumerates the different personalities, or distribution mechanisms, PolyORB provides.
- Chapter 4 [Building an application with PolyORB], page 13 presents the different steps to build a distributed application using PolyORB.
- Chapter 5 [Tasking model in PolyORB], page 19 details the use of tasking constructs within PolyORB.
- Chapter 6 [CORBA], page 25 describes PolyORB's implementation of OMG's CORBA.
- Chapter 7 [RT-CORBA], page 49 describes PolyORB's implementation of RT-CORBA, the real-time extensions of OMG's CORBA.
- Chapter 8 [DSA], page 53 describes PolyORB's implementation of the Ada Distributed Systems Annex.
- Chapter 9 [MOMA], page 95 describes PolyORB's implementation of MOMA, the Message Oriented Middleware for Ada.
- Chapter 10 [AWS], page 97 describes the integration of the Ada Web Server (AWS) framework into PolyORB.
- Chapter 11 [GIOP], page 99 describes PolyORB's implementation of GIOP, the protocol defined as part of CORBA.
- Chapter 12 [SOAP], page 107 describes PolyORB's implementation of SOAP.
- Chapter 13 [Tools], page 109 describes PolyORB's tools.
- Appendix B [Conformance to standards], page 113 discusses the conformance of the PolyORB's personalities to the CORBA and SOAP standards.
- Appendix C [References], page 117 provides a list of useful references to complete this documentation.
- Appendix D [GNU Free Documentation License], page 119 contains the text of the license under which this document is being distributed.

Conventions

Following are examples of the typographical and graphic conventions used in this guide:

- `Functions`, `utility program names`, `standard names`, and `classes`.
- '`Option flags`'
- '`File Names`', '`button names`', and '`field names`'.
- *Variables*.
- *Emphasis*.
- [optional information or parameters]
- Examples are described by text

```
and then shown this way.
```

Commands that are entered by the user are preceded in this manual by the characters "`$ `" (dollar sign followed by space). If your system uses this sequence as a prompt, then the commands will appear exactly as you see them in the manual. If your system uses some other prompt, then the command will appear with the `$` replaced by whatever prompt you are using.

Full file names are shown with the "`/`" character as the directory separator; e.g., '`parent-dir/subdir/myfile.adb`'. If you are using GNAT on a Windows platform, please note that the "`\`" character should be used instead.

1 Introduction to PolyORB

1.1 Introduction to distributed systems

A distributed system architecture comprises a network of computers and the software components that execute on those computers. Such architectures are commonly used to improve the performance, reliability, and reusability of complex applications. Typically, there is no shared address space available to remotely-located components (that is to say, components running on different nodes of the network), and therefore these components must communicate using some form of message-passing.

1.1.1 Using OS Network Services

There are several programming techniques for developing distributed applications. These applications have traditionally been developed using network programming interfaces such as sockets. Programmers have to perform explicit calls to operating system services, a task that can be tedious and error-prone. This includes initializing socket connections and determining peer location, marshalling and unmarshalling data structures, sending and receiving messages, debugging and testing several programs at the same time, and porting the application to several platforms to uncover subtle differences between various network interfaces.

Of course, this communication code can be encapsulated in wrappers to reduce its complexity, but it is clear that most of it can be automatically generated. Message passing diverts developer's attention from the application domain. The query and reply scenario is a classical scheme in distributed applications; using message passing for such a scheme can be compared to only using the "goto" mechanism in a non-distributed application. This is considered unacceptable methodology in modern software engineering. A cleaner and more structured approach consists in using subprograms.

In some respects, network programming can be compared to parallel programming. The user can decide to split his code into several pieces and to multiplex the execution of threads himself, using a table-driven model. The scheduling code ends up embedded in the user code. This solution is error-prone and fragile in regard to any future modification. Relying on an implementation of threads such as provided in a POSIX operating environment is a better solution. Relying on language primitives that support concurrency, such as Ada tasks, is best, as the underlying parallelism support is thus entirely abstracted.

1.1.2 Using a Middleware Environment

A middleware environment is intended to provide high level abstractions in order to easily develop user applications. Environments like CORBA or Distributed Computing Environment (DCE) provide a framework to develop client/server applications based on the Remote Procedure Call model (RPC). The RPC model is inspired by the query and reply scheme. In rough analogy to a regular procedure call, arguments are pushed onto a stream, along with some data specifying the remote procedure to be executed. The stream is transmitted over the network to the server. The server decodes the stream, performs the regular subprogram call locally, and then puts the output parameters into another stream, along with the exception (if any) raised by the subprogram execution. The server then sends this

stream back to the caller. The caller decodes the stream and raises the exception locally if needed.

CORBA provides the same enhancements to the remote procedure model that object-oriented languages provide to classical procedural languages. These enhancements include encapsulation, inheritance, type checking, and exceptions. These features are offered through an Interface Definition Language (IDL).

The middleware communication framework provides all the machinery to perform, somewhat transparently, remote procedure calls or remote object method invocations. For instance, each CORBA interface communicates through an Object Request Broker (ORB). A communication subsystem such as an ORB is intended to allow applications to use objects without being aware of their underlying message-passing implementation. In addition. the user may also require a number of more complex services to develop his distributed application. Some of these services are indispensable, for example a location service that allows clients to reference remote services via high level names (as opposed to a low level addressing scheme involving transport-specific endpoint addresses such as IP addresses and port numbers). Other services provide domain-independent interfaces that are frequently used by distributed applications.

If we return to the multi-threaded programming comparison, the middleware solution is close to what a POSIX library or a language like Esterel[1] would provide for developing concurrent applications. A middleware framework like DCE is close to a POSIX library in terms of abstraction levels. Functionalities are very low-level and very complex. CORBA is closer to Esterel in terms of development process. The control part of the application can be specified in a description language. The developer then has to fill in automatically generated source code templates (stubs and skeletons) to build the computational part of the application. The distribution is a pre-compilation process and the distributed boundaries are always explicit. Using CORBA, the distributed part is written in IDL and the core of the application is written in a host language such as C++.

1.1.3 Using a Distributed Language

Rather than defining a new language like the CORBA IDL, an alternative is to extend an existing programming language with distributed features. The distributed object paradigm provides a more object-oriented approach to programming distributed systems. The notion of a distributed object is an extension to the abstract data type that allows the services provided in the type interface to be called independently of where the actual service is executed. When combined with object-oriented features such as inheritance and polymorphism, distributed objects offer a more dynamic and structured computational environment for distributed applications.

The Distributed Systems Annex (DSA) of Ada defines several extensions that allow the user to write a distributed system entirely in Ada. The types of distributed objects, the services they provide, and the bodies of the remote methods to be executed are all defined in conventional Ada packages. The Ada model is analogous to the Java/RMI model. In both languages, the IDL is replaced by well-defined language constructs. Therefore, the language supports both remote procedure calls and remote object method invocations transparently, and the semantics of distribution are consistent with the rest of the language.

[1] `Esterel` is an imperative synchronous language designed for the specification and the development of reactive systems.

A program written in such a language is intended to communicate with a program written in the same language, but this apparent restriction has several useful consequences. The language can provide more powerful features because it is not constrained by the common features available in all host languages. In Ada, the user will define a specification of remote services and implement them exactly as he would for ordinary, non-distributed services. His Ada environment will compile them to produce a stub file (on the caller side) and a skeleton file that automatically includes the body of the services (on the receiver side). Creating objects, obtaining or registering object references or adapting the object skeleton to the user object implementation are made transparent because the language environment has a full control over the development process.

Comparing with multi-threaded programming once again, the language extension solution is equivalent to the solution adopted for tasking facilities in Ada. Writing a distributed application is as simple as writing a concurrent application: there is no binding consideration and no code to wrap. The language and its run-time system take care of most issues that would divert the programmer's attention from the application domain.

1.2 Distribution models and middleware standards

Middleware provides a framework that hides the complex issues of distribution, and offers the programmer high-level abstractions that allow easy and transparent construction of distributed applications. A number of different standards exist for creating object-oriented distributed applications. These standards define two subsystems that enable interaction between application partitions:

- the API seen by the developer's applicative objects;
- the protocol used by the middleware environment to interact with other nodes in the distributed application.

Middleware implementations also offer programming guidelines and development tools to ease the construction of large heterogeneous distributed systems. Many issues typical to distributed programming may still arise: application architectural choice, configuration or deployment. Since there is no "one size fits all" architecture, choosing the adequate distribution middleware in its most appropriate configuration is a key design point that dramatically impacts the design and performance of an application.

Consequently, applications need to rapidly tailor middleware to the specific distribution model they require. A distribution model is defined by the combination of distribution mechanisms made available to the application. Common examples of such mechanisms are Remote Procedure Call (RPC), Distributed Objects or Message Passing. A distribution infrastructure or middleware refers to software that supports one distribution model (or several), e.g.: OMG CORBA, Java Remote Method Invocation (RMI), the Distributed Systems Annex of Ada, Java Message Service (MOM).

1.3 The PolyORB generic middleware

Typical middleware implementations for one platform support only one set of such interfaces, predefined configuration capabilities and cannot interoperate with other platforms. In addition to traditional middleware implementations, PolyORB provides an original architecture to enable support for multiple interoperating distribution models in a uniform canvas.

PolyORB is a polymorphic, reusable infrastructure for building or prototyping new middleware adapted to specific application needs. It provides a set of components on top of which various instances can be elaborated. These instances (or personalities) are views on PolyORB facilities that are compliant to existing standards, either at the API level (application personality) or at the protocol level (protocol personality). These personalities are mutually exclusive views of the same architecture.

The decoupling of application and protocol personalities, and the support for multiple simultaneous personalities within the same running middleware, are key features required for the construction of interoperable distributed applications. This allows PolyORB to communicate with middleware that implements different distribution standards: PolyORB provides middleware-to-middleware interoperability (M2M).

PolyORB's modularity allows for easy extension and replacement of its core and personality components, in order to meet specific requirements. In this way, standard or application-specific personalities can be created in a streamlined process, from early stage prototyping to full-featured implementation. The PolyORB architecture also allows the automatic, just-in-time creation of proxies between incompatible environments.

You may find additional technical literature on PolyORB, including research papers and implementation notes, on the project websites: `http://libre.adacore.com/polyorb/` and `http://polyorb.objectweb.org/`.

Note: PolyORB is the project formerly known as DROOPI, a Distributed Reusable Object-Oriented Polymorphic Infrastructure

2 Installation

2.1 Supported Platforms

PolyORB has been compiled and successfully tested on the following platforms:

- AIX
- FreeBSD
- HP-UX
- Linux
- MacOS X
- Solaris
- Tru64
- VxWorks
- Windows

Note: PolyORB should compile and run on every target for which GNAT and the `GNAT.Sockets` *package are available.*

2.2 Build requirements

GNU tar is required to unpack PolyORB source packages.

Ada compiler:

- GNAT Pro 6.0 or later
- GNAT GPL 2007 or later
- FSF GCC 4.3 or later

For builds for cross targets, both a native and a cross compiler are required, as some tools (like an IDL-to-Ada compiler) are meant for use on the build host.

Optional:

- (Only for older versions of GNAT, and only if you want to build the CORBA application personality): A C++ compiler. The OMG IDL specification mandates that IDL source files be preprocessed according to standard C++ preprocessing rules. Newer versions of GNAT provide an integrated IDL preprocessor. This feature is detected and used automatically. However, for older versions of GNAT, PolyORB relies on an external preprocessor provided by a suitable C++ compiler. Please refer to the documentation of your particular version of GNAT to know if it supports this feature.
- XML/Ada (`http://libre.adacore.com/xmlada/`) if you want to build the SOAP protocol personality.

Note: per construction, the macro `configure` used to find your GNAT compiler looks first for the executable `gnatgcc`, then `adagcc` and finally `gcc` to find out which Ada compiler to use. You should be very careful with your path and executables if you have multiple GNAT versions installed. See the explanation below on the ADA environment variable if you need to override the default guess.

2.3 Build instructions

To compile and install PolyORB, execute:

```
$ ./configure [some options]
$ make
$ make install
```

This will install files in standard locations. If you want to choose a prefix other than '/usr/local', give configure a '--prefix=whereveryouwant' argument.

NOTE: you must use GNU make (version 3.80 or later) to build PolyORB.

2.4 Additional instructions for cross platforms

The 'RANLIB' environment variable must be set to the path of the cross 'ranlib' prior to running 'configure' with the appropriate --target option.

For example, for VxWorks 5 execute:

```
$ export RANLIB=ranlibppc
$ ./configure --target=powerpc-wrs-vxworks [some options]
$ make
$ make install
```

Only one PolyORB installation (native or cross) is currently possible with a given --prefix. If both a native and a cross installation are needed on the same machine, distinct prefixes must be used.

2.5 Building the documentation and PolyORB's examples

PolyORB's documentation and examples are built separately.

To build the examples, run make examples in the root directory. The build process will only build examples that correspond to the personalities you configured. Note that some examples require the CORBA COS Naming and IR services to be enabled (using --enable-corba-services="naming ir" on the configure command line).

Similarly, to build the documentation, run make docs.

You may install PolyORB's documentation in a standard location using make install.

2.6 Build Options

Available options for the 'configure' script include:

- '--with-appli-perso="..."': application personalities to build

  Available personalities: AWS, CORBA, DSA, MOMA

  e.g. '--with-appli-perso="corba moma"' to build both the CORBA and MOMA personalities
- '--with-proto-perso="..."': protocol personalities to build

  Available personalities: GIOP, SOAP

  e.g. '--with-proto-perso="giop soap"' to build both the GIOP and SOAP personalities
- '--with-idl-compiler="..."': select IDL compiler

  Available IDL compilers: iac (default), idlac

  e.g. '--with-idl-compiler=''iac''' to build iac

- '`--with-corba-services="..."`': CORBA COS services to build

  Available services: event, ir, naming, notification, time

  e.g. '`--with-corba-services="event naming"`' to build only COS Event and COS Naming.

By default, only the CORBA and GIOP personalities are built, and no CORBA Services are built.

- '`--with-openssl`': build SSL support and SSL dependent features, including the IIOP/SSLIOP personality
- '`--help`': list all options available
- '`--enable-shared`': build shared libraries.
- '`--enable-debug`': enable debugging information generation and supplementary runtime checks. Note that this option has a significant space and time cost, and is not recommended for production use.

2.7 Compiler, Tools and Run-Time libraries Options

The following environment variables can be used to override configure's guess at what compilers to use:

- `CC`: the C compiler
- `ADA`: the Ada compiler (e.g. gcc, gnatgcc or adagcc)
- `CXXCPP`, `CXXCPPFLAGS`: the preprocessor used by the IDL-to-Ada compiler (only when setting up the CORBA application personality). CORBA specifications require this preprocessor to be compatible with the preprocessing rules defined in the C++ programming language specifications.

For example, if you have two versions of GNAT installed and available in your `PATH`, and configure picks the wrong one, you can indicate what compiler should be used with the following (assuming Bourne shell syntax):

```
$ ADA=/path/to/good/compiler/gcc ./configure [options]
```

PolyORB will be compiled with GNAT build host's configuration, including run-time library. You may override this setting using `ADA_INCLUDE_PATH` and `ADA_OBJECTS_PATH` environment variables. See GNAT User's Guide for more details.

You can add specific build options to GNAT using the `EXTRA_GNATMAKE_FLAGS` variable:

```
$ EXTRA_GNATMAKE_FLAGS=--RTS=rts-sjlj ./configure [options]
```

You can also pass compiler-only flags using the `ADAFLAGS` variable.

NOTE: Developers building PolyORB from the version control repository will need to rebuild the configure script and other files. To do so, run the support/reconfig script after each update from the repository. In addition to the requirements above, developers will need autoconf 2.57 or newer, automake 1.6.3 or newer, and libtool 1.5.8 or newer.

2.8 Platform notes

Solaris (all versions):

/usr/ucb/tr is not suitable to build PolyORB. Your PATH must be set to that tr(1) is /usr/bin/tr, /use/xpg4/bin/tr or GNU tr.

Solaris 2.8:

/usr/bin/sed and /usr/ucb/sed will silently chop long lines, and /usr/xpg4/bin/sed will enter an endless loop while processing PolyORB files. GNU sed is required to configure and build PolyORB.

Tru64 5.1A:

The default maximal data segment size may not be sufficient to compile PolyORB. If a GNAT heap exhausted error message occurs during build, try raising this limit using:

```
ulimit -d unlimited
```

AIX 5.2:

PolyORB must be compiled with the -mminimal-toc compiler switch. This is taken care of automatically by the PolyORB configure script.

The 'ulimit' command may be needed as for Tru64 (see above).

HP-UX 11.00:

The version of install(1) from /opt/imake/bin on HP-UX is not suitable for installing PolyORB. Make sure that /opt/imake/bin is not on the PATH when building and installing PolyORB.

3 Overview of PolyORB personalities

A personality is an instantiation of specific PolyORB components. It provides the mechanisms specified by a distribution model, e.g. an API, a code generator or a protocol stack.

This section provides a brief overview of existing personalities.

Note: some of these personalities are available only through PolyORB's repository.

3.1 Application personalities

Application personalities constitute the adaptation layer between application components and middleware. They provide APIs and/or a code generator to register application entities with PolyORB's core, and interoperate with the core to allow the exchange of requests with remote entities.

3.1.1 CORBA

CORBA is the OMG specification of a Distributed Object Computing (DOC) distribution model (*[OMG04]*). It is now a well-known and well-established specification, used in a wide range of industrial applications.

PolyORB provides a CORBA-compatible implementation based on a mapping of the IDL language version 1.2 described in *[OMG01]* and CORBA core specifications. PolyORB also provides an implementation of various additional specifications described by the OMG, including `COS Services: COS Naming, Notification, Event, Time`, and additional specifications: `RT-CORBA`, `PortableInterceptors`, `DynamicAny`.

3.1.2 Distributed Systems Annex of Ada (DSA)

The Distributed Systems Annex of Ada (DSA) *[ISO06]* is a normative part of the language specification. It was first introduced in the "Ada 95" revision of the language (*[ISO95]*). It describes remote invocation schemes applied to most language constructs.

3.1.3 Message Oriented Middleware for Ada (MOMA)

MOMA (Message Oriented Middleware for Ada) provides message passing mechanisms. It is an Ada adaptation of Sun's Java Message Service (JMS) *[SUN99]*, a standardized API for common message passing models.

3.1.4 Ada Web Server (AWS)

The Web Server personality provides the same API as the Ada Web Server project (AWS) *[Obr03]*. It allows for the implementation of web services, web server applications, or classical web pages. AWS-based servers allow the programmer to directly interact with incoming or outgoing `HTTP` and `SOAP` requests.

3.2 Protocol personalities

Protocol personalities handle the mapping of requests (representing interactions between application entities) onto messages exchanged through a communication network, according to a specific protocol.

3.2.1 GIOP

GIOP is the transport layer of the CORBA specifications. GIOP is a generic protocol. This personality implements GIOP versions from 1.0 to 1.2 along with the CDR representation scheme to map data types between the neutral core layer and CDR streams. It also provides the following dedicated instances:

- IIOP supports synchronous request semantics over TCP/IP,
- IIOP/SSIOP supports synchronous request semantics using SSL sockets,
- MIOP instantiation of GIOP enables group communication over IP multicast,
- DIOP relies on UDP/IP communications to transmit one-way requests only.

3.2.2 SOAP

The SOAP protocol *[W3C03]* enables the exchange of structured and typed information between peers. It is a self-describing XML document *[W3C03]* that defines both its data and semantics. Basically, SOAP with `HTTP` bindings is used as a communication protocol for Web Services.

4 Building an application with PolyORB

4.1 Compile-time configuration

The user may configure some elements of a PolyORB application at compile-time.

4.1.1 Tasking runtimes

PolyORB provides several tasking runtimes. The user may select the most appropriate one, depending on application requirements. The tasking runtimes determine the constructs PolyORB may use for its internal synchronizations.

- `No_Tasking`: There is no dependency on the Ada tasking runtime, middleware is mono-task.
- `Full_Tasking`: Middleware uses Ada tasking constructs, middleware can be configured for multi-tasking.
- `Ravenscar` : Middleware uses Ada tasking constructs, with the limitations of the Ravenscar profile *[DB98]*. Middleware can be configured for multi-tasking.

See Chapter 5 [Tasking model in PolyORB], page 19 for more information on this point.

4.1.2 Middleware tasking policies

PolyORB provides several tasking policies. A tasking policy defines how tasks are used by the middleware to process incoming requests.

- `No_Tasking`: There is only one task in middleware, processing all requests.
- `Thread_Per_Session`: One task monitors communication entities. One task is spawned for each active connection. This task handles all incoming requests on this connection.
- `Thread_Per_Request`: One task monitors communication entities. One task is spawned for each incoming request.
- `Thread_Pool`: A set of tasks cooperate to handle all incoming requests.

See Chapter 5 [Tasking model in PolyORB], page 19 for more information on this point.

4.1.3 Sample files

PolyORB provides a set of predefined setup packages. You must 'with' one of them in your application node to activate the corresponding setup.

- `PolyORB.Setup.No_Tasking_Client`: a client node, without any tasking support, configured to use all protocol personalities built with PolyORB. Note that this configuration should not be used with multiple application tasks.
- `PolyORB.Setup.Thread_Pool_Client`: a client node, with tasking enabled, configured to use all protocol personalities built with PolyORB. This configuration places no restriction on the use of tasking by application code. Middleware tasking policy is `Thread_Pool`.
- `PolyORB.Setup.Ravenscar_TP_Server`: a server node, with tasking enabled, configured to use all protocol personalities built with PolyORB. Middleware tasking runtime follows Ravenscar's profile restrictions. Middleware tasking policy is `Thread_Pool`.

- `PolyORB.Setup.Thread_Per_Request_Server`: a server node, with tasking enabled, configured to use all protocol personalities built with PolyORB. Middleware tasking policy is `Thread_Per_Request`.
- `PolyORB.Setup.Thread_Per_Session_Server`: a server node, with tasking enabled, configured to use all protocol personalities built with PolyORB. Middleware tasking policy is `Thread_Per_Session`.
- `PolyORB.Setup.Thread_Pool_Server`: a server node, with tasking enabled, configured to use all protocol personalities built with PolyORB. Middleware tasking policy is `Thread_Pool`.

To use one of these configurations, add a dependency on one of these packages, for example, `with PolyORB.Setup.Thread_Pool_Server;`. The elaboration of the application (based on Ada rules) and the initialization of the partition (based on the application personalities mechanisms) will properly set up your application.

4.2 Run-time configuration

The user may configure some elements of a PolyORB application at run time.

Using the default configurations provided by PolyORB, the parameters are read in the following order: command line, environment variables, configuration file. PolyORB will use the first value that matches the searched parameter.

4.2.1 Using a configuration file

A configuration file may be used to configure a PolyORB node. A sample configuration file may be found in '`src/polyorb.conf`'.

The syntax of the configuration file is:

- empty lines and lines that have a '#' in column 1 are ignored;
- sections can be started by lines of the form `[ SECTION-NAME ]`;
- variable assignments can be performed by lines of the form `VARIABLE-NAME = VALUE`.

  Any variable assignment is local to a section.

  Assignments that occur before the first section declaration are relative to section [environment]. Section and variable names are case sensitive.

  Furthermore, each time a value starts with `"file:"`, the contents of the file are used instead.

Default search path for '`polyorb.conf`' is current directory. Environment variable `POLYORB_CONF` may be used to set up information on configuration file.

PolyORB's configuration file allows the user to

1. enable/disable the output of debug information
2. set up default reference on naming service
3. select the default protocol personality
4. set up each protocol personality

The configuration file is read once when running a node, during initialization. Look in the sample configuration file '`src/polyorb.conf`' to see the available sections and variables.

4.2.2 Using environment variables

A variable `Var.Iable` in section `[Sec]` can be overridden by setting environment variable `"POLYORB_SEC_VAR_IABLE"`.

4.2.3 Using the command line

PolyORB allows to set up configuration variables on the command line. The syntax is close to the one described in configuration files. A variable `Var.Iable` in section `[Sec]` can be overridden with flag `--polyorb-<sec>-<var>-<iable>[=<value>]`.

4.2.4 Using a source file

Many embedded systems do not have a filesystem or a shell, so the previous run-time configuration methods cannot be used on these targets. On these platforms, a PolyORB node can also be configured using the API of package `PolyORB.Parameters.Static`. An example configuration file may be found in '`examples/static/po_static_conf.ads`'.

An array of PolyORB parameters of type `Static_Parameters_Array` is first declared containing a list of pairs of Variable and Value strings. The syntax is close to the one described in configuration files. A variable `Var.Iable` in section `[Sec]` is specified as the pair of strings `"[sec]var.iable", "<value>"`.

There is no need to with this '`po_static_conf.ads`' in the application source code, the only requirement is that the array is exported with the external name `"__polyorbconf_optional"`. This allows to modify PolyORB parameters without recompiling the application, just relinking it. For example:

```
$ gnatmake -c po_static_conf.ads `polyorb-config`
$ gnatmake -b -l server.adb `polyorb-config` -largs po_static_conf.o
```

Note the `-l` flag to gnatmake for linking only, and the need to specify to the linker the object file with the array using `-largs` if no package withs it.

It should be noticed that this static array of parameters is read at elaboration time only, this API cannot be used to modify the PolyORB configuration at run-time.

4.3 Setting up protocol personalities

PolyORB allows the user to activate some of the available protocol personalities and to set up the preferred protocol. Protocol-specific parameters are defined in their respective sections.

4.3.1 Activating/Deactivating protocol personalities

Protocol activation is controlled by PolyORB's configuration file.

The section `[access_points]` controls the initialization of *access points*. An access point is a node entry point that may serve incoming requests.

```
[access_points]
soap=enable
iiop=enable
diop=disable
uipmc=disable
```

This example activates SOAP and IIOP, but deactivates DIOP and MIOP.

The section `[modules]` controls the activation/deactivation of some modules within PolyORB. It is used to enable *bindings* to remote entities.

```
[modules]
binding_data.soap=enable
binding_data.iiop=enable
binding_data.diop=disable
binding_data.uipmc=disable
```

This example enables the creation of bindings to remote objects using SOAP or IIOP. Objects cannot be reached using DIOP or UIPMC.

Note: by default, all configured personalities are activated.

4.3.2 Configuring protocol personality preferences

The user may affect a *preference* to each protocol personality. The protocol with the higher preference will be selected among possible protocols to send a request to a remote node.

See polyorb.binding_data.<protocol>.preference in section [protocol] to set up protocol's preference.

Possible protocols are defined as the protocols available on the remote node, as advertised in its *object reference*. IOR or corbaloc references may support multiple protocols; URI references support only one protocol.

Each protocol supports a variety of configuration parameters, please refer to the protocols' sections for more details.

4.4 Activating debugging traces

To activate the output of debug information, you must first configure and compile PolyORB with debugging traces activated (which is the default, unless your build is configured with --enable-debug-policy=ignore).

To output debugging traces on a selected package, create a configuration file with a [log] section and the name of the packages for which you want debug information:

```
# Sample configuration file, output debug for PolyORB.A_Package
[log]
polyorb.a_package=debug
```

Note that some packages may not provide such information. See the sample configuration file 'src/polyorb.conf' for the complete list of packages that provide traces.

A default logging level may be specified using a line of the form

```
default=<level>
```

Time stamps may optionally be prepended to every generated trace. This is enabled using:

```
timestamp=true
```

4.5 Tracing exceptions

To trace exception propagation in PolyORB's source code, activate debugging traces for package PolyORB.Exceptions.

4.6 polyorb.gpr

This section describes how to build your program using project files. An alternative method, using polyorb-config, is described in the following section. polyorb-config is intended primarily for Unix-like systems. The project-file method will work on all supported systems.

To build your application, create a project file as usual. Import the `polyorb.gpr` project by putting `with "polyorb";` in your project file.

Set the ADA_PROJECT_PATH environment variable to point to the directory containing `polyorb.cfg`, which is `<prefix>/lib/gnat`. If SOAP is being used, ADA_PROJECT_PATH must also be set so we can find `xmlada.gpr`.

If your project file is `my_proj.gpr`, you can build it by saying:

```
$ gnatmake -P my_proj
```

See the GNAT User's Guide and the GNAT Reference Manual for more information on project files.

4.7 polyorb-config

`polyorb-config` returns path and library information on PolyORB's installation. It can be used on the `gnatmake` command line, like this:

```
$ gnatmake my_program.adb ‘polyorb-config‘
```

```
NAME
       polyorb-config  - script to get information about the installed version
       of PolyORB.

SYNOPSIS
       polyorb-config [--prefix[=DIR]] [--exec-prefix[=DIR]] [--version|-v]
       [--config] [--libs] [--cflags] [--idls] [--help]

DESCRIPTION
       polyorb-config  is  a  tool  that is used to determine the compiler and
       linker flags that should be used to compile and link programs that  use
       PolyORB.

OPTIONS
       polyorb-config accepts the following options:

       --prefix[=DIR]
               Output the directory in which PolyORB architecture-independent
              files are installed, or set this directory to DIR.

       --exec-prefix[=DIR]
               Output the directory in which  PolyORB  architecture-dependent
              files are installed, or set this directory to DIR.

       --version
              Print  the  currently installed version of PolyORB on the stan-
              dard output.

       --config
              Print the configuration of the currently installed  version  of
              PolyORB on the standard output.

       --libs Print  the  linker  flags  that are necessary to link a PolyORB
              program.

       --cflags
              Print the compiler flags that are necessary to compile a  Poly-
```

```
        ORB program.

--idls
        Output flags to set up path to CORBA's IDL for idlac.

--with-appli-perso=P,P,P
        Restrict output to only those flags relevant to the listed
        applicative personalities.

--with-proto-perso=P,P,P
        Restrict output to only those flags relevant to the listed
        protocol personalities.

--with-corba-services=S,S,S
       Restrict output to only those flags relevant to the listed
       services.

--help  Print help message.
```

5 Tasking model in PolyORB

5.1 PolyORB Tasking runtimes

PolyORB may use any of three different tasking runtimes to manage and synchronize tasks, if any. Tasking runtime capabilities are defined in the Ada Reference Manual *[ISO06]*.

The choice of a specific tasking runtime is a compile-time parameter, Section 4.1.1 [Tasking runtimes], page 13 for more details on their configuration.

5.1.1 Full tasking runtime

Full tasking runtime refers to the configuration in which there are dependencies on the tasking constructs defined in chapter 9 of *[ISO06]*. It makes use of all capabilities defined in this section to manage and synchronize tasks.

In this configuration, a PolyORB application must be compiled and linked with a tasking-capable Ada runtime.

5.1.2 No tasking runtime

No tasking runtime refers to the configuration in which there is no dependency on tasking constructs. Thus, no tasking is required.

In this configuration, a PolyORB application may be compiled and linked with a tasking-capable Ada runtime or a no-tasking Ada runtime.

5.1.3 Ravenscar tasking runtime

Ravenscar tasking runtime refers to the configuration in which tasking constructs are compliant with the *Ravenscar tasking restricted profile*.

In this configuration, a PolyORB application may be compiled and linked with a tasking-capable Ada runtime or a Ravenscar Ada runtime.

To configure tasking constructs used by PolyORB, one must instantiate the `PolyORB.Setup.Tasking.Ravenscar` generic package shown below to set up tasks and protected objects used by PolyORB core.

```
------------------------------------------------------------------------------
--                                                                          --
--                           POLYORB COMPONENTS                             --
--                                                                          --
--     P O L Y O R B . S E T U P . T A S K I N G . R A V E N S C A R      --
--                                                                          --
--                                 S p e c                                  --
--                                                                          --
--         Copyright (C) 2002-2007, Free Software Foundation, Inc.          --
--                                                                          --
-- PolyORB is free software; you  can  redistribute  it and/or modify it    --
-- under terms of the  GNU General Public License as published by the  Free --
-- Software Foundation;  either version 2,  or (at your option)  any  later --
-- version. PolyORB is distributed  in the hope that it will be  useful,    --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details.  You should have received  a copy of the GNU  --
```

```
-- General Public License distributed with PolyORB; see file COPYING. If    --
-- not, write to the Free Software Foundation, 59 Temple Place - Suite 330, --
-- Boston, MA 02111-1307, USA.                                              --
--                                                                          --
-- As a special exception,  if other files  instantiate  generics from this --
-- unit, or you link  this unit with other files  to produce an executable, --
-- this  unit  does not  by itself cause  the resulting  executable  to  be --
-- covered  by the  GNU  General  Public  License.  This exception does not --
-- however invalidate  any other reasons why  the executable file  might be --
-- covered by the  GNU Public License.                                      --
--                                                                          --
--               PolyORB is maintained by ACT Europe.                       --
--                   (email: sales@act-europe.fr)                           --
--                                                                          --
------------------------------------------------------------------------------

--  You should instantiate this package to set up a ravenscar profile.

with System;

with PolyORB.Tasking.Profiles.Ravenscar.Threads.Annotations;

with PolyORB.Tasking.Profiles.Ravenscar.Threads;
with PolyORB.Tasking.Profiles.Ravenscar.Mutexes;
with PolyORB.Tasking.Profiles.Ravenscar.Condition_Variables;

generic
   Number_Of_Application_Tasks    : Integer;
   --  Number of tasks created by the user.

   Number_Of_System_Tasks         : Integer;
   --  Number of tasks created by the PolyORB run-time library.

   Number_Of_Conditions           : Integer;
   --  Number of preallocated conditions.

   Number_Of_Mutexes              : Integer;
   --  Number of preallocated mutexes.

   Task_Priority                  : System.Priority;
   --  Priority of the tasks of the pool.

   Storage_Size                   : Natural;
   --  Stack size of the system tasks.

package PolyORB.Setup.Tasking.Ravenscar is

   package Threads_Package is
      new PolyORB.Tasking.Profiles.Ravenscar.Threads
     (Number_Of_Application_Tasks,
      Number_Of_System_Tasks,
      Task_Priority,
      Storage_Size);

   package Thread_Annotations_Package is new Threads_Package.Annotations;

   package Conditions_Package is
      new PolyORB.Tasking.Profiles.Ravenscar.Condition_Variables
```

```
       (Threads_Package,
        Number_Of_Conditions);

     package Mutexes_Package is
        new PolyORB.Tasking.Profiles.Ravenscar.Mutexes
       (Threads_Package,
        Number_Of_Mutexes);

  end PolyORB.Setup.Tasking.Ravenscar;
```

5.2 PolyORB ORB Tasking policies

PolyORB ORB Tasking policies control the creation of tasks to process all middleware internal jobs, e.g. request processing, I/O monitoring.

Note: there is a dependency between ORB Tasking policies, and the runtime used, as detailed below.

5.2.1 No Tasking

Under the No Tasking ORB policy, no tasks are created within the middleware instance: it uses the environment task to process all jobs. Note that this policy is not thread safe and is compatible with the No tasking runtime only.

5.2.2 Thread Pool

Under the Thread Pool ORB policy, the middleware creates a pool of threads during initialization of PolyORB. This pool processes all jobs. The number of tasks in the thread pool can be configured by three parameters in the `[tasking]` configuration section.

- `min_spare_threads` indicates the number of tasks created at startup.
- `max_spare_threads` is a ceiling. When a remote subprogram call is completed, its anonymous task is deallocated if the number of unused tasks already in the pool is greater than the ceiling. If not, then the task is queued in the pool.
- `max_threads` indicates the maximum number of tasks in the pool.

See Section 5.3 [PolyORB Tasking configuration], page 21, for more information on how to configure the number of tasks in the thread pool.

5.2.3 Thread Per Session

Under the Thread Per Session ORB policy, the middleware creates one task when a new session (one active connection) is opened. The task terminates when the session is closed.

5.2.4 Thread Per Request

Under the Thread Per Request ORB policy, the middleware creates one task per incoming request. The task terminates when the request is completed.

5.3 PolyORB Tasking configuration

The following parameters allow the user to set up some of the tasking parameters.

```
###############################################################################
```

```
# Parameters for tasking
#

[tasking]
# Default storage size for all threads spawned by PolyORB
#storage_size=262144

# Number of threads by Thread Pool tasking policy
#min_spare_threads=4
#max_spare_threads=4
#max_threads=4
```

5.4 PolyORB ORB Controller policies

The PolyORB ORB Controller policies are responsible for the management of the global state of the middleware: they assign middleware internal jobs, or I/Os monitoring to middleware tasks.

ORB Controller policies grant access to middleware internals and affect one action for each middleware task. They ensure that all tasks work concurrently in a thread-safe manner.

5.4.1 No Tasking

The No Tasking ORB Controller policy is dedicated to no-tasking middleware configurations; the middleware task executes the following loop: process internal jobs, then monitor I/Os.

5.4.2 Workers

The Workers ORB Controller policy is a simple controller policy: all tasks are equal, they may alternatively and randomly process requests or wait for I/O sources.

Note: this is the default configuration provided by PolyORB sample setup files, See *Section 4.1.3 [Sample files], page 13*.

5.4.3 Half Sync/Half Async

The Half Sync/Half Async ORB Controller policy implements the "Half Sync/Half Async" design pattern: it discriminates between one thread dedicated to I/O monitoring that queue middleware jobs; another pool of threads dequeue jobs and process them.

Note: this pattern is well-suited to process computation-intensive requests.

5.4.4 Leader/Followers

The Leader/Followers ORB Controller policy implements the "Leader/Followers " design pattern: multiple tasks take turns to monitor I/O sources and then process requests that occur on the event sources.

Note: this pattern is adapted to process a lot of light requests.

5.5 PolyORB ORB Controller configuration

The following parameters allow the user to set up parameters for ORB Controllers.

```
###############################################################################
```

```
# Parameters for ORB Controllers
#

[orb_controller]
# Interval between two polling actions on one monitor
#polyorb.orb_controller.polling_interval=0

# Timeout when polling on one monitor
#polyorb.orb_controller.polling_timeout=0
```

6 CORBA

6.1 What you should know before Reading this section

This section assumes that the reader is familiar with the CORBA specifications described in *[OMG04]* and the *IDL-to-Ada* mapping defined in *[OMG01]*.

6.2 Installing CORBA application personality

Ensure PolyORB has been configured and then compiled with the CORBA application personality. See Chapter 4 [Building an application with PolyORB], page 13 for more details on how to check installed personalities.

To build the CORBA application personality, see Chapter 2 [Installation], page 7.

6.3 IDL-to-Ada compiler

PolyORB provides two IDL-to-Ada compilers:

1. `iac` is the new, optimized PolyORB IDL-to-Ada compiler.
2. `idlac` is the legacy PolyORB IDL-to-Ada compiler,

6.3.1 Usage of `iac`

`iac` is PolyORB's new IDL-to-Ada compiler. It supports many command line parameters to control code generation optimizations such as use of static hashing for deterministic request dispatching, and optimized GIOP marshalling for CORBA applications.

```
NAME
       iac - PolyORB's IDL-to-Ada compiler

SYNOPSIS
       iac [options] file [-cppargs args...]

DESCRIPTION
       iac is an IDL-to-Ada compiler, compliant with version 1.2 of the
       ``Ada Language Mapping Specification'' produced by the OMG.

OPTIONS
       iac accepts the following options:


  -h       Print this help message, and do nothing else

  file is the name of the .idl file (.idl suffix optional)

  -E       Preprocess only
  -k       Keep temporary files
  -o DIR   Output directory (DIR must exist)
  -p       Produce source on standard output
  -q       Quiet mode

  -dm      Generate debug messages when analyzing scopes
```

```
   -df      Dump the frontend tree (the IDL tree)

   -cppargs Pass arguments to the C++ preprocessor
   -I <dir> Shortcut -cppargs -I directory. Use this flag
            for the imported entities
   -nocpp   Do not preprocess input

   -gnatW8  Use UTF-8 character encoding in Ada output.
            (Default is Latin-1.)

   -<lang>  Generate code for one of the following languages:

    types   Generate a list of all types present in the IDL file
            -p       Print the list generated

    ada     (default) Generate Ada source code
            -i       Generate implementation packages
            -c       Generate code for client side only
            -s       Generate code for server side only
            -d       Generate delegation package (defunct)
            -ir      Generate code for interface repository
            -noir    Do not generate code for interface repository (default)
            -hc      Minimize CPU time in perfect hash tables in skels
            -hm      Minimize memory use in perfect hash tables in skels
                     This is the default.
            -rs      Use the SII/SSI to handle requests
            -rd      Use the DII/DSI to handle requests (default)
            -da      Dump the Ada tree
            -db      Generate only the package bodies
            -ds      Generate only the package specs
            -dw      Output the withed entities
            -dt      Output tree warnings
            -di      Generate code for imported entities

    idl     Dump parsed IDL file
            -b n     Base to output integer literals
                     As a default (zero) use base from input
            -e       Expand IDL Tree
            -df      Dump IDL Tree (may be used in conjunction with -e
                     to dump the expanded IDL tree)
            -di      Output IDL code of imported entities  (may be
                     used in conjunction with -e to output the
                     expanded IDL code)

 EXIT STATUS
        iac returns one of the following values upon exit:

        0      Successful completion

        1      Usage error

        2      Illegal IDL specification
```

`iac` creates several files :

- `myinterface.ads`, `myinterface.adb` : these files contain the mapping for user defined types (client and server side).
- `myinterface-impl.ads`, `myinterface-impl.adb` : these files are to be filled in by the

user. They contain the implementation of the server. They are generated only if the -i flag is specified.

- `myinterface.ads`, `myinterface.adb` : these files contain the client stubs for the interface.
- `myinterface-skel.ads`, `myinterface-skel.adb` : these files contain the server-side skeletons for the interface.
- `myinterface-helper.ads`, `myinterface-helper.adb` : these files contain subprograms to marshal data into CORBA Any containers.
- `myinterface-ir_info.ads`, `myinterface-ir_info.adb` : these files contain code for registering IDL definitions in the CORBA Interface Repository. They are generated only if the `'-ir'` flag is specified.
- `myinterface-cdr.ads`, `myinterface-cdr.adb` : these files contain code for optimized CDR marshalling of GIOP messages. They are generated only if the `'-rs'` flag is specified.

6.3.2 Usage of `idlac`

`idlac` is PolyORB's IDL-to-Ada compiler.

```
NAME
       idlac - PolyORB's IDL-to-Ada compiler

SYNOPSIS
       idlac [-Edikpqv] [-[no]ir] [-gnatW8] [-o DIR] idl_file [-cppargs ...]

DESCRIPTION
       idlac is an IDL-to-Ada compiler, compliant with version 1.2 of the
       ``Ada Language Mapping Specification'' produced by the OMG.

OPTIONS
       idlac accepts the following options:

       -E      Preprocess only.

       -d      Generate delegation package.

       -i      Generate implementation template.

       -s      Generate server side code.

       -c      Generate client side code.

       -k      Keep temporary files.

       -p      Produce source on standard output.

       -q      Be quiet (default).

       -v      Be verbose.

       -ir     Generate code for interface repository.

       -noir   Don't generate code for interface repository (default).
```

```
        -gnatW8
               Use UTF8 character encoding

        -o DIR Specify output directory

        -cppargs ARGS
               Pass ARGS to the C++ preprocessor.

        -I dir  Shortcut for -cppargs -I dir.

 EXIT STATUS
        idlac returns one of the following values upon exit:

        0      Successful completion

        1      Usage error

        2      Illegal IDL specification
```

`idlac` creates several files :

- `myinterface.ads`, `myinterface.adb` : these files contain the mapping for user defined types (client and server side).
- `myinterface-impl.ads`, `myinterface-impl.adb` : these files are to be filled in by the user. They contain the implementation of the server. They are generated only if the -i flag is specified.
- `myinterface.ads`, `myinterface.adb` : these files contain the client stubs for the interface.
- `myinterface-skel.ads`, `myinterface-skel.adb` : these files contain the server-side skeletons for the interface.
- `myinterface-helper.ads`, `myinterface-helper.adb` : these files contain subprograms to marshal data into CORBA Any containers.
- `myinterface-ir_info.ads`, `myinterface-ir_info.adb` : these files contain code for registering IDL definitions in the CORBA Interface Repository. They are generated only if the `'-ir'` flag is specified.

6.3.3 Difference between idlac and iac

This section lists the main differences between `idlac` and `iac`

- `iac` is backward compatible with `idlac`, but lacks the following feature:
  1. generation of delegation files.

`iac` implements additional name clash resolution rules. When the name of an IDL operation clashes with a primitive operation of Ada.Finalization.Controlled (of which CORBA.Object.Ref is a derived type), it is prefixed with `"IDL_"` in generated sources.

6.4 Resolving names in a CORBA application

PolyORB implements the CORBA COS Naming service.

6.4.1 po_cos_naming

po_cos_naming is a standalone server that supports the CORBA COS Naming specification. When launched, it returns its IOR and corbaloc, which can then be used by other CORBA applications.

If you want po_cos_naming to return the same IOR or corbaloc at each startup, you must set a default listen port for the protocol personalities you use. See Section 4.3.2 [Configuring protocol personality preferences], page 16 for more details.

po_cos_naming can output its IOR directly to a file using the -file <filename> flag. This, in conjonction with the 'file://' naming scheme provided by CORBA, provides a convenient way to store initial references to the Naming Service.

```
Usage: po_cos_naming
 -file <filename> : output COS Naming IOR to 'filename'
 -help : print this help
 [PolyORB command line configuration variables]
```

6.4.2 Registering the reference to the COS Naming server

You have two ways to register the reference to the root context of the COS Naming server the application will use:

- Setting up the name_service entry in the [corba] section in your configuration file, name_service is the IOR or corbaloc of the COS Naming server to use. See Section 4.2.1 [Using a configuration file], page 14 for more details.
- Registering an initial reference using the -ORBInitRef NamingService=<IOR> or -ORBInitRef NamingService=<corbaloc> command-line argument. See the CORBA specifications for more details.
- Registering an initial reference for NamingService using the CORBA.ORB.Register_Initial_Reference function. See the CORBA specifications for more details.

6.4.3 Using the COS Naming

PolyORB provides a helper package to manipulate the COS Naming in your applications. See Section 6.9 [PolyORB specific APIs], page 42 for more details.

6.5 The CORBA Interface Repository

PolyORB implements the CORBA Interface Repository.

6.5.1 po_ir

po_ir is a standalone server that supports the CORBA Interface Repository. When launched, it returns its IOR and corbaloc, which can then be used by other CORBA applications.

If you want po_ir to return the same IOR or corbaloc at each startup, you must set a default listen port for the protocol personalities you use. See Section 4.3.2 [Configuring protocol personality preferences], page 16 for more details.

6.5.2 Using the Interface Repository

The IDL-to-Ada compiler generates a helper package that allows you to register all entities defined in your IDL specification in the Interface Repository.

6.6 Building a CORBA application with PolyORB

6.6.1 echo example

We consider building a simple "Echo" CORBA server and client. This application echoes a string. The source code for this example is located in the '`examples/corba/echo`' directory in the PolyORB distribution. This applications uses only basic elements of CORBA.

To build this application, you need the following pieces of code:

1. IDL definition of an `echo` object
2. Implementation code for the `echo` object
3. Code for client and server nodes

6.6.1.1 IDL definition of an echo object

This interface defines an `echo` object with a unique method `echoString`. Per construction, this method returns its argument.

```
interface Echo {
  string echoString (in string Mesg);
};
```

6.6.1.2 Implementation code for the echo object

Package `Echo.Impl` is an implementation of this interface. This implementation follows the *IDL-to-Ada* mapping.

```
------------------------------------------------------------------------------
--                                                                          --
--                           POLYORB COMPONENTS                             --
--                                                                          --
--                            E C H O . I M P L                             --
--                                                                          --
--                                 S p e c                                  --
--                                                                          --
--         Copyright (C) 2002-2008, Free Software Foundation, Inc.          --
--                                                                          --
-- PolyORB is free software; you  can  redistribute  it and/or modify it    --
-- under terms of the  GNU General Public License as published by the  Free --
-- Software Foundation;  either version 2,  or (at your option)  any  later --
-- version. PolyORB is distributed  in the hope that it will be  useful,    --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details.  You should have received  a copy of the GNU  --
-- General Public License distributed with PolyORB; see file COPYING. If    --
-- not, write to the Free Software Foundation, 59 Temple Place - Suite 330, --
-- Boston, MA 02111-1307, USA.                                              --
--                                                                          --
-- As a special exception,  if other files  instantiate  generics from this --
-- unit, or you link  this unit with other files  to produce an executable, --
-- this  unit  does not  by itself cause  the resulting  executable  to  be --
-- covered  by the  GNU  General  Public  License.  This exception does not --
-- however invalidate  any other reasons why  the executable file  might be --
```

```
-- covered by the  GNU Public License.                                      --
--                                                                          --
--                PolyORB is maintained by ACT Europe.                      --
--                    (email: sales@act-europe.fr)                          --
--                                                                          --
------------------------------------------------------------------------------

with CORBA;
with PortableServer;

package Echo.Impl is

   --  My own implementation of echo object.
   --  This is simply used to define the operations.

   type Object is new PortableServer.Servant_Base with null record;

   type Object_Acc is access Object;

   function EchoString
     (Self : access Object;
      Mesg : CORBA.String)
     return CORBA.String;

end Echo.Impl;



------------------------------------------------------------------------------
--                                                                          --
--                           POLYORB COMPONENTS                             --
--                                                                          --
--                            E C H O . I M P L                             --
--                                                                          --
--                                 B o d y                                  --
--                                                                          --
--         Copyright (C) 2002-2008, Free Software Foundation, Inc.          --
--                                                                          --
-- PolyORB is free software; you  can  redistribute  it and/or modify it    --
-- under terms of the  GNU General Public License as published by the  Free --
-- Software Foundation;  either version 2,  or (at your option)  any  later --
-- version. PolyORB is distributed  in the hope that it will be  useful,    --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details.  You should have received  a copy of the GNU  --
-- General Public License distributed with PolyORB; see file COPYING. If    --
-- not, write to the Free Software Foundation, 59 Temple Place - Suite 330, --
-- Boston, MA 02111-1307, USA.                                              --
--                                                                          --
-- As a special exception,  if other files  instantiate  generics from this --
-- unit, or you link  this unit with other files  to produce an executable, --
-- this  unit  does not  by itself cause  the resulting  executable  to  be --
-- covered  by the  GNU  General  Public  License.  This exception does not --
-- however invalidate  any other reasons why  the executable file  might be --
-- covered by the  GNU Public License.                                      --
--                                                                          --
--                PolyORB is maintained by ACT Europe.                      --
--                    (email: sales@act-europe.fr)                          --
--                                                                          --
```

```
------------------------------------------------------------------------------

with Ada.Text_IO;

with Echo.Skel;
pragma Warnings (Off, Echo.Skel);
--  No entity from Echo.Skel is referenced.

package body Echo.Impl is

   ----------------
   -- EchoString --
   ----------------

   function EchoString
     (Self : access Object;
      Mesg : CORBA.String)
     return CORBA.String
   is
      pragma Warnings (Off);
      pragma Unreferenced (Self);
      pragma Warnings (On);
   begin
      Ada.Text_IO.Put_Line
        ("Echoing string:  " & CORBA.To_Standard_String (Mesg)
         & " ");
      return Mesg;
   end EchoString;

end Echo.Impl;
```

Note: `Echo.Impl` *body requires a dependency on* `Echo.Skel` *to ensure the elaboration of skeleton code and the correct setup of PolyORB's internals.*

6.6.1.3 Test code for client and server nodes

Client and server code demonstrate how to make a remote invocation on a CORBA object, and how to set up an object on a server node.

Note: the dependency on `PolyORB.Setup.Client` *or* `PolyORB.Setup.No_Tasking_Server` *enforces compile-time configuration, see* *Section 4.1.3 [Sample files], page 13.*

- Client code tests a simple remote invocation on an object. It is a no-tasking client. A reference to the object is built from a stringified reference (or `IOR`), which is passed on command line.

```
------------------------------------------------------------------------------
--                                                                          --
--                           POLYORB COMPONENTS                             --
--                                                                          --
--                               C L I E N T                                --
--                                                                          --
--                                 B o d y                                  --
--                                                                          --
--         Copyright (C) 2002-2004 Free Software Foundation, Inc.           --
--                                                                          --
-- PolyORB is free software; you  can  redistribute  it and/or modify it    --
-- under terms of the  GNU General Public License as published by the  Free --
```

```
-- Software Foundation;  either version 2,  or (at your option)  any  later --
-- version. PolyORB is distributed  in the hope that it will be  useful,    --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details.  You should have received  a copy of the GNU  --
-- General Public License distributed with PolyORB; see file COPYING. If    --
-- not, write to the Free Software Foundation, 59 Temple Place - Suite 330, --
-- Boston, MA 02111-1307, USA.                                              --
--                                                                          --
-- As a special exception,  if other files  instantiate  generics from this --
-- unit, or you link  this unit with other files  to produce an executable, --
-- this  unit  does not  by itself cause  the resulting  executable  to  be --
-- covered  by the  GNU  General  Public  License.  This exception does not --
-- however invalidate  any other reasons why  the executable file  might be --
-- covered by the  GNU Public License.                                      --
--                                                                          --
--               PolyORB is maintained by ACT Europe.                       --
--                   (email: sales@act-europe.fr)                           --
--                                                                          --
------------------------------------------------------------------------------

--   echo client.

with Ada.Command_Line;
with Ada.Text_IO;
with CORBA.ORB;

with Echo;

with PolyORB.Setup.Client;
pragma Warnings (Off, PolyORB.Setup.Client);

with PolyORB.Utils.Report;

procedure Client is
   use Ada.Command_Line;
   use Ada.Text_IO;
   use PolyORB.Utils.Report;

   Sent_Msg, Rcvd_Msg : CORBA.String;
   myecho : Echo.Ref;

begin
   New_Test ("Echo client");

   CORBA.ORB.Initialize ("ORB");
   if Argument_Count /= 1 then
      Put_Line ("usage : client <IOR_string_from_server>|-i");
      return;
   end if;

   --  Getting the CORBA.Object

   CORBA.ORB.String_To_Object
     (CORBA.To_CORBA_String (Ada.Command_Line.Argument (1)), myecho);

   --  Checking if it worked
```

```
      if Echo.Is_Nil (myecho) then
         Put_Line ("main : cannot invoke on a nil reference");
         return;
      end if;

      --  Sending message

      Sent_Msg := CORBA.To_CORBA_String (Standard.String'("Hello Ada !"));
      Rcvd_Msg := Echo.echoString (myecho, Sent_Msg);

      --  Printing result

      Put_Line ("I said : " & CORBA.To_Standard_String (Sent_Msg));
      Put_Line ("The object answered : " & CORBA.To_Standard_String (Rcvd_Msg));

      End_Report;

   exception
      when E : CORBA.Transient =>
         declare
            Memb : CORBA.System_Exception_Members;
         begin
            CORBA.Get_Members (E, Memb);
            Put ("received exception transient, minor");
            Put (CORBA.Unsigned_Long'Image (Memb.Minor));
            Put (", completion status: ");
            Put_Line (CORBA.Completion_Status'Image (Memb.Completed));

            End_Report;
         end;
   end Client;
```

- The server code sets up a no-tasking node. The object is registered to the `RootPOA`. Then an `IOR` reference is built to enable interaction with other nodes.

```
------------------------------------------------------------------------------
--                                                                          --
--                           POLYORB COMPONENTS                             --
--                                                                          --
--                               S E R V E R                                --
--                                                                          --
--                                 B o d y                                  --
--                                                                          --
--         Copyright (C) 2002-2007, Free Software Foundation, Inc.          --
--                                                                          --
-- PolyORB is free software; you  can  redistribute  it and/or modify it    --
-- under terms of the  GNU General Public License as published by the  Free --
-- Software Foundation;  either version 2,  or (at your option)  any  later --
-- version. PolyORB is distributed  in the hope that it will be  useful,    --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details.  You should have received  a copy of the GNU  --
-- General Public License distributed with PolyORB; see file COPYING. If    --
-- not, write to the Free Software Foundation, 51 Franklin Street, Fifth    --
-- Floor, Boston, MA 02111-1301, USA.                                       --
--                                                                          --
-- As a special exception,  if other files  instantiate  generics from this --
```

```
-- unit, or you link  this unit with other files  to produce an executable, --
-- this  unit  does not  by itself cause  the resulting  executable  to  be --
-- covered  by the  GNU  General  Public  License.  This exception does not --
-- however invalidate  any other reasons why  the executable file  might be --
-- covered by the  GNU Public License.                                      --
--                                                                          --
--                  PolyORB is maintained by AdaCore                        --
--                     (email: sales@adacore.com)                           --
--                                                                          --
------------------------------------------------------------------------------

with Ada.Text_IO;

with CORBA.Impl;
with CORBA.Object;
with CORBA.ORB;

with PortableServer.POA.Helper;
with PortableServer.POAManager;

with Echo.Impl;

with PolyORB.CORBA_P.CORBALOC;

--  Setup server node: use no tasking default configuration

with PolyORB.Setup.No_Tasking_Server;
pragma Warnings (Off, PolyORB.Setup.No_Tasking_Server);

procedure Server is
begin

   declare
      Argv : CORBA.ORB.Arg_List := CORBA.ORB.Command_Line_Arguments;

   begin
      CORBA.ORB.Init (CORBA.ORB.To_CORBA_String ("ORB"), Argv);

      declare
         Root_POA : PortableServer.POA.Local_Ref;

         Ref : CORBA.Object.Ref;

         Obj : constant CORBA.Impl.Object_Ptr := new Echo.Impl.Object;

      begin

         --  Retrieve Root POA

         Root_POA := PortableServer.POA.Helper.To_Local_Ref
           (CORBA.ORB.Resolve_Initial_References
            (CORBA.ORB.To_CORBA_String ("RootPOA")));

         PortableServer.POAManager.Activate
           (PortableServer.POA.Get_The_POAManager (Root_POA));

         --  Set up new object
```

```
         Ref := PortableServer.POA.Servant_To_Reference
           (Root_POA, PortableServer.Servant (Obj));

         --  Output IOR

         Ada.Text_IO.Put_Line
           ("'"
            & CORBA.To_Standard_String (CORBA.Object.Object_To_String (Ref))
            & "'");
         Ada.Text_IO.New_Line;

         --  Output corbaloc

         Ada.Text_IO.Put_Line
           ("'"
            & CORBA.To_Standard_String
            (PolyORB.CORBA_P.CORBALOC.Object_To_Corbaloc (Ref))
            & "'");

         --  Launch the server

         CORBA.ORB.Run;
      end;
   end;
end Server;
```

6.6.1.4 Compilation and execution

To compile this demo,

1. Process the IDL file with `idlac` (or `iac`)

   ```
   $ idlac echo.idl
   ```

2. Compile the client node

   ```
   $ gnatmake client.adb `polyorb-config`
   ```

3. Compile the server node

   ```
   $ gnatmake server.adb `polyorb-config`
   ```

Note the use of backticks (`). This means that `polyorb-config` is first executed, and then the command line is replaced with the output of the script, setting up library and include paths and library names.

To run this demo:

- run '`server`', the server outputs its IOR, a hexadecimal string with the IOR: prefix:

  ```
  $ ./server
  Loading configuration from polyorb.conf
  No polyorb.conf configuration file.
  'IOR:01534f410d00000049444c3[..]'
  ```

- In another shell, run '`client`', passing cut-and-pasting the complete IOR on the command line:

  ```
  $ ./client 'IOR:01534f410d00000049444c3[..]'
  Echoing string: " Hello Ada ! "
  I said : Hello Ada !
  The object answered : Hello Ada !
  ```

6.6.2 Other examples

PolyORB provides other examples to test other CORBA features. These examples are located in the 'example/corba' directory in the PolyORB distribution.

- 'all_functions' tests CORBA parameter passing modes (in, out, ..);
- 'all_types' tests CORBA types;
- 'echo' is a simple CORBA demo;
- 'random' is a random number generator;
- 'send' tests MIOP specific API.

6.7 Configuring a CORBA application

To configure a CORBA application, you need to separately configure PolyORB and the GIOP protocol (or any other protocol personality you wish to use).

6.7.1 Configuring PolyORB

Please refer to Chapter 4 [Building an application with PolyORB], page 13 for more information on PolyORB's configuration.

6.7.2 Configuring GIOP protocol stack for PolyORB

The GIOP protocol is separated from the CORBA application personality. See Section 11.3 [Configuring the GIOP personality], page 99 for more information on GIOP's configuration.

6.7.3 Configuring Security services for PolyORB

PolyORB provides support for some elements of the CORBA Security mechanisms. This sections lists the corresponding configuration parameters.

6.7.3.1 Supported mechasnisms

PolyORB provides support for the following security mechanisms:

1. SSL/TLS protected transport;
2. GSSUP (user/password) authentication mechanism;
3. identity assertion and backward trust evaluation.

6.7.3.2 Compile-time configuration

To enable security support, applications must 'with' one of the predefined setup packages:

1. PolyORB.Setup.Secure_Client - for client side support only;
2. PolyORB.Setup.Secure_Server - for both client and server side support.

6.7.3.3 Run-time configuration

1. Capsule configuration

   This section details the configuration parameters for capsule configuration.

   ```
   [security_manager]
   # List of sections for configure client's credentials
   #own_credentials=my_credentials
   #
   ```

```
# Client requires integrity proteced messages
#integrity_required=true
#
# Client requires confiodentiality protected messages
#confidentiality_required=true
#
# Client requires security association to detect replay (not supported
for now)
#detect_replay_required=true
#
# Client requires security association to detect message sequence
errors (not
# supported for now)
#detect_misordering_required=true
#
# Client requires target authentication
#establish_trust_in_target_required=true
#
# Client requires client authentication (usually not applicable at
all)
#establish_trust_in_client_required=true
#
# (rare useful)
#identity_assertion_required=true
#
# (rare useful)
#delegation_by_client_required=true
```

2. Credentials configuration

   This section details configuration parameters for defining a program's credentials. Depending on the mechanisms used for the transport and authentication layers, the credentials configuration section may define configuration only for one transport mechanism and/or one authentication mechanism.

```
#[my_credentials]
#
# TLS protected transport mechanism used as transport mechanism
#transport_credentials_type=tls
#
# Connection method. Available methods: tls1, ssl3, ssl2
#tls.method=tls1
#
# Certificate file name
#tls.certificate_file=my.crt
#
# Certificate chain file name
#tls.certificate_chain_file=
#
# Private key file name
#tls.private_key_file=my.key
#
# Name of file, at which CA certificates for verification purposes are
#located
#tls.certificate_authority_file=root.crt
#
# Name of directory, at which CA certificates for verification
#purposes are
# located
```

```
#tls.certificate_authority_path=
#
# List of available ciphers
#tls.ciphers=ALL
#
# Verify peer certificate
#tls.verify_peer=true
#
# Fail if client don't provide ceritificate (server only)
#tls.verify_fail_if_no_peer_certificate=true
#
# GSSUP (user/password) mechanism as authentication mechanism
#authentication_credentials_type=gssup
#
# User name
#gssup.username=username@domain
#
# User password
#gssup.password=password
#
# Target name for which user/password pair is applicable
#gssup.target_name=@domain
```

3. POA configuration

   This section details configuration parameters for defining security characteristics of objects managed by POA. The POA's name is used as the section name.

```
#[MySecurePOA]
#
# Unprotected invocations is allowed
#unprotected_invocation_allowed=true
#
# Section name for configuration of used protected transport mechanism
#(if any)
#transport_mechanism=tlsiop
#
# Section name for configuration of used authentication mechanism (if
#any)
#authentication_mechanism=my_gssup
#
# Target require client authentication at authentication layer (in
#addition
# to authentication at transport layer)
#authentication_required=true
#
# Name of file for backward trust evalutation rules
#backward_trust_rules_file=file.btr
#
# Section name for configuration of authorization tokens authority
#privilege_authorities=
```

4. TLS protected transport mechanism configuration

   This section details configuration parameters for the TLS protected transport mechanism. The section name for mechanism configuration is defined in the POA configuration.

```
[tlsiop]
# List of access points
#addresses=127.0.0.1:3456
```

5. GSSUP authentication mechanism

   This section details configuration parameters for the GSSUP authentication mechanism. The section name for mechanism configuration is defined in the POA configuration.

```
#[my_gssup]
#
# Authentication mechanism
#mechanism=gssup
#
# Target name
#gssup.target_name=@domain
#
# User name/password mapping file
#gssup.passwd_file=passwd.pwd
```

6.7.4 Command line arguments

The CORBA specifications define a mechanism to pass command line arguments to your application, using the `CORBA::ORB:Init` method.

For now, PolyORB supports the following list of arguments:

- `InitRef` to pass initial reference.

6.8 Implementation Notes

PolyORB strives to support CORBA specifications as closely as possible. However, on rare occasions, the implementation adapts the specifications to actually enable its completion. This section provides information on the various modifications we made.

6.8.1 Tasking

PolyORB provides support for tasking and no-tasking, using configuration parameters. Please refer to Chapter 4 [Building an application with PolyORB], page 13 for more information on PolyORB's configuration.

When selecting a tasking-capable runtime, ORB-related functions are thread safe, following the IDL-to-Ada mapping recommendations.

6.8.2 Implementation of CORBA specifications

In some cases, the CORBA specifications do not describe the semantics of the interface in sufficient detail. We add an `Implementation Notes` tag to the package specification to indicate the modifications or enhancements we made to the standard.

In some cases, the IDL-to-Ada mapping specifications and the CORBA specifications conflict. We add an `Implementation Notes` tag to the package specification to indicate this issue. Whenever possible, PolyORB follows the CORBA specifications.

6.8.3 Additions to the CORBA specifications

In some cases, the specifications lack features that may be useful. We add an `Implementation Notes` tag to the package specification to detail the additions we made to the standard.

In addition to the above, PolyORB follows some of the recommendations derived from the OMG Issues for Ada 2003 Revision Task Force mailing list (see http://www.omg.org/issues/ada-rtf.html for more information).

6.8.4 Interface repository

The documentation of the PolyORB's CORBA Interface Repository will appear in a future revision of PolyORB.

6.8.5 Policy Domain Managers

You have two ways to register the reference to the CORBA Policy Domain Manager the application will use:

- Setting up the policy_domain_manager entry in the [corba] section in your configuration file, policy_domain_manager is the IOR or corbaloc of the COS Naming server to use. See Section 4.2.1 [Using a configuration file], page 14 for more details.
- Registering an initial reference using the -ORB InitRef PolyORBPolicyDomainManager=<IOR> or -ORB InitRef PolyORBPolicyDomainManager=<corbaloc> command-line argument. See the CORBA specifications for more details.
- Registering an initial reference for PolyORBPolicyDomainManager using the CORBA.ORB.Register_Initial_Reference function. See the CORBA specifications for more details.

6.8.6 Mapping of exceptions

For each exception defined in the CORBA specifications, PolyORB provides the Raise_<excp_name> function, a utility function that raises the exception <excp_name>, along with its exception member. PolyORB also defines the Get_Members function (as defined in the IDL-to-Ada mapping) to provide accessors to retrieve information on the exception.

In addition, for each exception defined in a user-defined IDL specification, the IDL-to-Ada compiler will generate a Raise_<excp_name> function in the Helper package. It is a utility function that raises the exception <excp_name>, along with its exception member.

6.8.7 Additional information to CORBA::Unknown

When a CORBA application raises an Ada exception that is not part of the IDL specifications, nor defined by the CORBA specifications, then this exception is translated into a CORBA::UNKNOWN exception.

To help debugging CORBA applications, PolyORB supports a specific service context to the GIOP protocol personality that conveys exception information. When displaying exception information, server-side specific exception information is delimited by *"<Invocation Exception Info: ...>"*

Here is an example from the all_types example provided by PolyORB.

```
Exception name: CORBA.UNKNOWN
Message: 4F4D0001M
<Invocation Exception Info: Exception name: CONSTRAINT_ERROR
Message: all_types-impl.adb:315 explicit raise
Call stack traceback locations:
0x84d279c 0x84c1e78 0x84b92c6 0x84b8e9>
Call stack traceback locations:
```

```
0x81d0425 0x81d0554 0x81d6d8c 0x81fd02b 0x81fc091 0x82eea12 0x83e4c22 0x807b69a 0xb7a15e3e
```

Note that call stack tracebacks can be translated into symbolic form using the `addr2line` utility that comes with GNAT.

6.8.8 Internals packages

PolyORB sometimes declares internal types and routines inside CORBA packages. These entities are gathered into an `Internals` child package. You should not use these functions: they are not portable, and may be changed in future releases.

6.9 PolyORB's specific APIs

PolyORB defines packages to help in the development of CORBA programs.

- Section 6.9.1 [PolyORB.CORBA_P.CORBALOC], page 43:

  This package defines a helper function to build a `corbaloc` stringified reference from a CORBA object reference.
- Section 6.9.2 [PolyORB.CORBA_P.Naming_Tools], page 44:

  This package defines helper functions to ease interaction with CORBA COS Naming.
- Section 6.9.3 [PolyORB.CORBA_P.Server_Tools], page 46:

  This package defines helper functions to ease set up of a simple CORBA Server.

6.9.1 PolyORB.CORBA_P.CORBALOC

```
------------------------------------------------------------------------------
--                                                                          --
--                           POLYORB COMPONENTS                             --
--                                                                          --
--              P O L Y O R B . C O R B A _ P . C O R B A L O C             --
--                                                                          --
--                                 S p e c                                  --
--                                                                          --
--           Copyright (C) 2004-2006, Free Software Foundation, Inc.        --
--                                                                          --
-- PolyORB is free software; you  can  redistribute  it and/or modify it    --
-- under terms of the  GNU General Public License as published by the  Free --
-- Software Foundation;  either version 2,  or (at your option)  any  later --
-- version. PolyORB is distributed  in the hope that it will be  useful,    --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details.  You should have received  a copy of the GNU  --
-- General Public License distributed with PolyORB; see file COPYING. If    --
-- not, write to the Free Software Foundation, 51 Franklin Street, Fifth    --
-- Floor, Boston, MA 02111-1301, USA.                                       --
--                                                                          --
-- As a special exception,  if other files  instantiate  generics from this --
-- unit, or you link  this unit with other files  to produce an executable, --
-- this  unit  does not  by itself cause  the resulting  executable  to  be --
-- covered  by the  GNU  General  Public  License.  This exception does not --
-- however invalidate  any other reasons why  the executable file  might be --
-- covered by the  GNU Public License.                                      --
--                                                                          --
--                  PolyORB is maintained by AdaCore                        --
--                     (email: sales@adacore.com)                           --
--                                                                          --
------------------------------------------------------------------------------

--  Helper functions to manage CORBA corbaloc references

with CORBA.Object;

package PolyORB.CORBA_P.CORBALOC is

   function Object_To_Corbaloc
     (Obj : CORBA.Object.Ref'Class)
     return CORBA.String;
   --  Convert reference to corbaloc, return corbaloc of best profile

end PolyORB.CORBA_P.CORBALOC;
```

6.9.2 PolyORB.CORBA_P.Naming_Tools

```
------------------------------------------------------------------------------
--                                                                          --
--                           POLYORB COMPONENTS                             --
--                                                                          --
--         P O L Y O R B . C O R B A _ P . N A M I N G _ T O O L S          --
--                                                                          --
--                                 S p e c                                  --
--                                                                          --
--         Copyright (C) 2001-2008, Free Software Foundation, Inc.          --
--                                                                          --
-- PolyORB is free software; you  can  redistribute  it and/or modify it    --
-- under terms of the  GNU General Public License as published by the  Free --
-- Software Foundation;  either version 2,  or (at your option)  any  later --
-- version. PolyORB is distributed  in the hope that it will be  useful,    --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details.  You should have received  a copy of the GNU  --
-- General Public License distributed with PolyORB; see file COPYING. If    --
-- not, write to the Free Software Foundation, 51 Franklin Street, Fifth    --
-- Floor, Boston, MA 02111-1301, USA.                                       --
--                                                                          --
-- As a special exception,  if other files  instantiate  generics from this --
-- unit, or you link  this unit with other files  to produce an executable, --
-- this  unit  does not  by itself cause  the resulting  executable  to  be --
-- covered  by the  GNU  General  Public  License.  This exception does not --
-- however invalidate  any other reasons why  the executable file  might be --
-- covered by the  GNU Public License.                                      --
--                                                                          --
--                  PolyORB is maintained by AdaCore                        --
--                     (email: sales@adacore.com)                           --
--                                                                          --
------------------------------------------------------------------------------

--  Wrappers for the COS Naming service to facilitate retrievel of object
--  references by IOR or by name.

with Ada.Finalization;

with CORBA.Object;
with CosNaming.NamingContext;

package PolyORB.CORBA_P.Naming_Tools is

   function Locate (Name : CosNaming.Name) return CORBA.Object.Ref;

   function Locate
     (Context : CosNaming.NamingContext.Ref;
      Name    : CosNaming.Name) return CORBA.Object.Ref;
   --  Locate an object given its name, given as an array of name components.

   function Locate
     (IOR_Or_Name : String; Sep : Character := '/') return CORBA.Object.Ref;

   function Locate
     (Context     : CosNaming.NamingContext.Ref;
      IOR_Or_Name : String;
```

```
      Sep         : Character := '/') return CORBA.Object.Ref;
   --  Locate an object by IOR or name. If the string does not start with
   --  "IOR:", the name will be parsed before it is looked up, using
   --  Parse_Name below.

   procedure Register
     (Name   : String;
      Ref    : CORBA.Object.Ref;
      Rebind : Boolean := False;
      Sep    : Character := '/');
   --  Register an object by its name by binding or rebinding.
   --  The name will be parsed by Parse_Name below; any necessary contexts
   --  will be created on the name server.
   --  If Rebind is True, then a rebind will be performed if the name
   --  is already bound.

   procedure Unregister (Name : String);
   --  Unregister an object by its name by unbinding it

   type Server_Guard is limited private;
   procedure Register
     (Guard  : in out Server_Guard;
      Name   : String;
      Ref    : CORBA.Object.Ref;
      Rebind : Boolean := False;
      Sep    : Character := '/');
   --  A Server_Guard object is an object which is able to register a server
   --  reference in a naming service (see Register above), and destroy this
   --  name using Unregister when the object disappears (the program terminates
   --  or the Server_Guard object lifetime has expired).

   function Parse_Name
     (Name : String;
      Sep : Character := '/') return CosNaming.Name;
   --  Split a sequence of name component specifications separated with Sep
   --  characters into a name component array. Any leading Sep is ignored.

private
   --  implementation removed
end PolyORB.CORBA_P.Naming_Tools;
```

6.9.3 PolyORB.CORBA_P.Server_Tools

```
------------------------------------------------------------------------------
--                                                                          --
--                           POLYORB COMPONENTS                             --
--                                                                          --
--           P O L Y O R B . C O R B A _ P . S E R V E R _ T O O L S        --
--                                                                          --
--                                 S p e c                                  --
--                                                                          --
--           Copyright (C) 2001-2009, Free Software Foundation, Inc.        --
--                                                                          --
-- PolyORB is free software; you  can  redistribute  it and/or modify it    --
-- under terms of the  GNU General Public License as published by the  Free --
-- Software Foundation;  either version 2,  or (at your option)  any  later --
-- version. PolyORB is distributed  in the hope that it will be  useful,    --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details.  You should have received  a copy of the GNU  --
-- General Public License distributed with PolyORB; see file COPYING. If    --
-- not, write to the Free Software Foundation, 51 Franklin Street, Fifth    --
-- Floor, Boston, MA 02111-1301, USA.                                       --
--                                                                          --
-- As a special exception,  if other files  instantiate  generics from this --
-- unit, or you link  this unit with other files  to produce an executable, --
-- this  unit  does not  by itself cause  the resulting  executable  to  be --
-- covered  by the  GNU  General  Public  License.  This exception does not --
-- however invalidate  any other reasons why  the executable file  might be --
-- covered by the  GNU Public License.                                      --
--                                                                          --
--                  PolyORB is maintained by AdaCore                        --
--                     (email: sales@adacore.com)                           --
--                                                                          --
------------------------------------------------------------------------------

--  Helper functions for CORBA servers. Note that using this unit implies
--  using the Portable Object Adapter.

with CORBA.Object;
with PortableServer.POA;

package PolyORB.CORBA_P.Server_Tools is

   pragma Elaborate_Body;

   type Hook_Type is access procedure;
   Initiate_Server_Hook : Hook_Type;
   --  Access to a procedure to be called upon start up.
   --  See Initiate_Server for more details.

   procedure Activate_Server;
   --  Start a new ORB, and initialize the Root POA.
   --
   --  If the Initiate_Server_Hook variable is not null, the designated
   --  procedure will be called after initializing the ORB.

   procedure Initiate_Server (Start_New_Task : Boolean := False);
   --  Calls Activate_Server then starts ORB main loop.
```

```
   --  If Start_New_Task is True, a new task will be created and control will
   --  be returned to the caller. Otherwise, the ORB main loop will be executed
   --  in the current context.

   function Get_Root_POA return PortableServer.POA.Local_Ref;
   --  Return the Root_POA attached to the current ORB instance.

   procedure Initiate_Servant
     (S : PortableServer.Servant;
      R : out CORBA.Object.Ref'Class);
   --  Initiate a servant: register a servant to the Root POA.
   --  If the Root POA has not been initialized, initialize it.

   procedure Reference_To_Servant
     (R : CORBA.Object.Ref'Class;
      S : out PortableServer.Servant);
   --  Convert a CORBA.Object.Ref into a PortableServer.Servant.

   procedure Servant_To_Reference
     (S : PortableServer.Servant;
      R : out CORBA.Object.Ref'Class);
   --  Convert a PortableServer.Servant into CORBA.Object.Ref.

   procedure Initiate_Well_Known_Service
     (S    : PortableServer.Servant;
      Name : String;
      R    : out CORBA.Object.Ref'Class);
   --  Make S accessible through a reference appropriate for
   --  generation of a corbaloc URI with a named key of Name.

end PolyORB.CORBA_P.Server_Tools;
```

7 RT-CORBA

7.1 What you should know before Reading this section

This section assumes that the reader is familiar with the Real-Time CORBA specifications described in *[OMG02a]* and *[OMG03]*.

7.2 Installing RT-CORBA

The RT-CORBA library is installed as part of the installation of the CORBA personality. Note that you may have to select specific run-time options to enable full compliance with RT-CORBA specifications and ensure real time behavior.

7.3 Configuring RT-CORBA

This section details how to configure your application to use the RT-CORBA library.

7.3.1 `PolyORB.RTCORBA_P.Setup`

The RT-CORBA specifications mandate that the implementation provide a mechanism to set up some of its internals.

The package `PolyORB.RTCORBA_P.Setup` provides an API to set up the `PriorityMapping` and `PriorityTransform` objects.

```
------------------------------------------------------------------------------
--                                                                          --
--                           POLYORB COMPONENTS                             --
--                                                                          --
--             P O L Y O R B . R T C O R B A _ P . S E T U P                --
--                                                                          --
--                                 S p e c                                  --
--                                                                          --
--         Copyright (C) 2003-2004 Free Software Foundation, Inc.           --
--                                                                          --
-- PolyORB is free software; you  can  redistribute  it and/or modify it    --
-- under terms of the  GNU General Public License as published by the  Free --
-- Software Foundation;  either version 2,  or (at your option)  any  later --
-- version. PolyORB is distributed  in the hope that it will be  useful,    --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details.  You should have received  a copy of the GNU  --
-- General Public License distributed with PolyORB; see file COPYING. If    --
-- not, write to the Free Software Foundation, 59 Temple Place - Suite 330, --
-- Boston, MA 02111-1307, USA.                                              --
--                                                                          --
-- As a special exception,  if other files  instantiate  generics from this --
-- unit, or you link  this unit with other files  to produce an executable, --
-- this  unit  does not  by itself cause  the resulting  executable  to  be --
-- covered  by the  GNU  General  Public  License.  This exception does not --
-- however invalidate  any other reasons why  the executable file  might be --
-- covered by the  GNU Public License.                                      --
--                                                                          --
--                PolyORB is maintained by ACT Europe.                      --
--                    (email: sales@act-europe.fr)                          --
```

```
--                                                                          --
------------------------------------------------------------------------------

--  Implementation Notes: RTCORBA specifications defines objects that
--  are (Ada) programming language objects rather than CORBA
--  objects. Therefore the normal mechanism for coupling an
--  implementation to the code that uses it (an object reference) does
--  not apply. The implementation must provide specific mechanisms to
--  enable this coupling.
--
--  This package provides accessors to configure them. It supports the
--  following objects:
--  * PriorityMapping
--  * PriorityTransform

with RTCORBA.PriorityMapping;
with RTCORBA.PriorityTransform;

package PolyORB.RTCORBA_P.Setup is

   --  PriorityMapping

   type PriorityMapping_Access is
     access all RTCORBA.PriorityMapping.Object'Class;

   procedure Set_Priority_Mapping
     (Mapping : RTCORBA.PriorityMapping.Object'Class);
   pragma Inline (Set_Priority_Mapping);
   --  Set RT-ORB PriorityMapping object,
   --  overrides previous settings, if any.

   function Get_Priority_Mapping return PriorityMapping_Access;
   pragma Inline (Get_Priority_Mapping);
   --  Return RT-ORB PriorityMapping object.

   --  PriorityTransform

   type PriorityTransform_Access is
     access all RTCORBA.PriorityTransform.Object'Class;

   procedure Set_Priority_Transform
     (Transform : RTCORBA.PriorityTransform.Object'Class);
   pragma Inline (Set_Priority_Transform);
   --  Set RT-ORB global Priority Mapping object,
   --  overrides previous settings, if any.

   function Get_Priority_Transform return PriorityTransform_Access;
   pragma Inline (Get_Priority_Transform);
   --  Return RT-ORB global Priority Mapping object.

end PolyORB.RTCORBA_P.Setup;
```

7.4 RTCORBA.PriorityMapping

PolyORB provides different implementations of this specification:

- `RTCORBA.PriorityMapping.Direct` maps CORBA priorities directly to native priori-

ties. If the CORBA priority is not in `System.Priority'Range`, then the mapping is not possible.

- `RTCORBA.PriorityMapping.Linear` maps each individual native priority to a contiguous range of CORBA priorities, so that the complete CORBA priority range is used up for the mapping. See '`rtcorba-prioritymapping-linear.adb`' for more details.

7.5 RTCosScheduling Service

7.5.1 Overview

PolyORB provides an implementation of the RTCosScheduling service defined in *[OMG02a]*.

PolyORB uses some permissions stated in the specifications to allow for easy configuration of `ClientScheduler` and `ServerScheduler`, defined in the following sections.

Additional information on the use of the API may be found in the RTCosScheduling example in '`examples/corba/rtcorba/rtcosscheduling`'.

7.5.2 `RTCosScheduling::ClientScheduler`

Client side *activities* are defined in a configuration file that can be loaded using '`RTCosScheduling.ClientScheduler.Impl.Load_Configuration_File`'

On the client side, the user can set up

- current task priority, using registered `PriorityMapping` object.

This file has the following syntax, derived from PolyORB configuration file syntax:

```
# Name of the activity
[activity activity1]

# Activity priority, in RTCORBA.Priority'Range
priority=10000
```

In this example, activity `activity1` is defined with priority `10'000`.

7.5.3 `RTCosScheduling::ServerScheduler`

Server side *POAs* and *objects* are defined in a configuration file that can be loaded using '`RTCosScheduling.ClientScheduler.Impl.Load_Configuration_File`'

On the server side, the user can set up

- object priority, using registered `PriorityMapping` object.
- all RT-CORBA-specific POA configuration parameters.

This file has the following syntax, derived from PolyORB configuration file syntax:

```
# Name of the object
[object object1]

# Object priority, in RTCORBA.Priority'Range
priority=10000
```

In this example, object `object1` is defined with priority `10'000`.

```
# Name of the POA
[poa poa1]

# PriorityModelPolicy for POA
priority_model=CLIENT_PROPAGATED
default_priority=0 # not meaningful for CLIENT_PROPAGATED

# Threadpools attached to POA
threadpool_id=1

# Name of the POA
[poa poa2]

# PriorityModelPolicy for POA
priority_model=SERVER_DECLARED
default_priority=40

# Threadpools attached to POA
threadpool_id=2

# Name of the POA
[poa poa3]

# POA with no defined policies
```

In this example, Two POAs are defined: POA `poa1` will use the `CLIENT_PROPAGATED` PriorityModel Policy, default value is not meaningful for this configuration, `poa1` will use the Threadpool #1; POA `poa2` will use the `SERVER_DECLARED` PriorityModel Policy, default server priority is 40, `poa2` will use the Threadpool #2. Note that both policies are optional and can be omitted.

8 Ada Distributed Systems Annex (DSA)

8.1 Introduction to the Ada DSA

A critical feature of the Distributed Systems Annex (DSA) is that it allows the user to develop his application the same way whether this application is going to be executed as several programs on a distributed system, or as a single program on a non-distributed system. The DSA has been designed to minimize the source changes needed to convert an ordinary non-distributed program into a distributed program.

The simplest way to start with DSA is to develop the application on a non-distributed system. Of course, the design of the application should take into account the fact that some units are going to be accessed remotely. In order to write a distributed Ada program, it is necessary for the user to label by means of categorization pragmas some of library level compilation units of the application program. The units that require categorization are typically those that are called remotely, and those that provide the types used in remote invocations.

In order to ensure that distributed execution is possible, these units are restricted to contain only a limited set of Ada constructs. For instance, if the distributed system has no shared memory, shared variables must be forbidden. To specify the nature of these restrictions, the DSA provides several categorization pragmas, each of which excludes some language constructs from the categorized package.

Of course, the user can develop the non-distributed application with his usual software engineering environment. It is critical to note that the user needs no specialized tools to develop his/her distributed application. For instance, he can debug his application with the usual debugger. Note that a non-distributed program is not to be confused with a distributed application composed of only one program. The latter is built with the help of the configuration tool and includes the communication library.

Once the non-distributed version of the program is complete, it has to be configured into separate partitions. This step is surprisingly simple, compared to that of developing the application itself. The configuration step consists of mapping sets of compilation units into individual partitions, and specifying the mapping between partitions and nodes in the computer network. This mapping is specified and managed by means of a gnatdist configuration.

The distributed version of the user application should work as is, but even when a program can be built both as a non-distributed or a distributed program using the same source code, there may still be differences in program execution between the distributed and non-distributed versions. These differences are discussed in subsequent sections (see Section 8.1.7.4 [Pragma Asynchronous], page 65 and Section 8.1.7.5 [Pragma All_Calls_Remote], page 67).

Developing a non-distributed application in order to distribute it later is the natural approach for a novice. Of course, it is not always possible to write a distributed application as a non-distributed application. For instance, a client/server application does not belong to this category because several instances of the client can be active at the same time. It is very easy to develop such an application using PolyORB; we shall describe how to do this in the following sections.

8.1.1 Architecture of a Distributed Ada Application

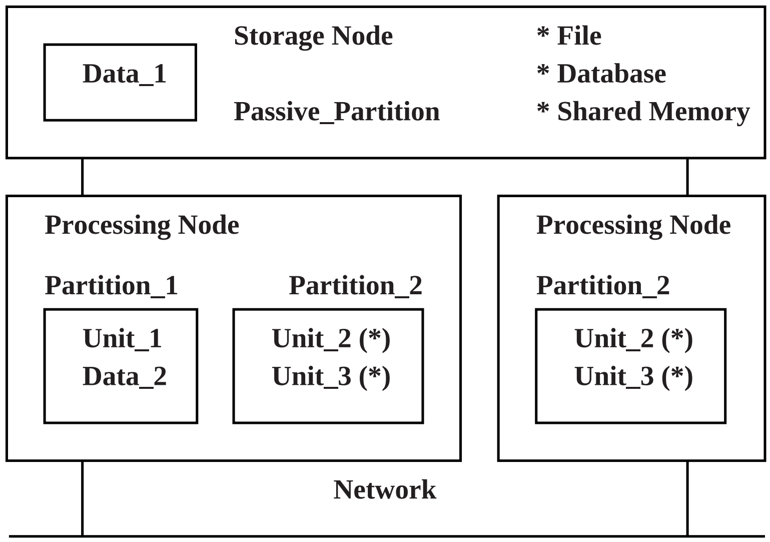
A distributed system is an interconnection of one or more processing nodes and zero or more storage nodes. A distributed program comprises one or more partitions. A partition is an aggregate of library units. Partitions communicate through shared data or RPCs. A passive partition has no thread of control. Only a passive partition can be configured on a storage node. An active partition has zero or more threads of control and has to be configured on a processing node.

The library unit is the core component of a distributed Ada application. The user can explicitly assign library units to a partition. Partitioning is a post-compilation process. The user identifies interface packages at compile-time. These packages are categorized using pragmas. Each of these pragmas supports the use of one of the following classical paradigms:

- Remote subprograms: For the programmer, a remote subprogram call is similar to a regular subprogram call. Run-time binding using access-to-subprogram types can also be used with remote subprograms. These remote subprograms are declared in library units categorized as remote call interface (RCI).
- Distributed objects: Special-purpose access types can designate remote objects. When a primitive dispatching operation is invoked on an object designated by such a remote access, a remote call is performed transparently on the partition on which the object resides. The types of these distributed objects are declared in library units categorized as remote types (RT).
- Shared objects: Global data can be shared among active partitions, providing a repository similar to shared memory, a shared file system or a database. Entryless protected objects allow safe concurrent access and update of shared objects. This feature is orthogonal to the notion of distributed objects, which are only accessed through exported services. These shared objects are declared in library units categorized as shared passive (SP).

The remotely-called subprograms declared in a library unit categorized as remote call interface (RCI) or remote types (RT) may be either statically or dynamically bound. The partition on which a statically bound remote subprogram is executed can be determined before the call. This is a static remote subprogram call. In contrast, a remote method or a dereference of an access to remote subprogram are dynamically bound remote calls, because the partition on which the remote subprogram is executed is determined at runtime, by the actuals of the call.

In the following example, Data_1 and Data_2 are shared passive (SP) library units. Data_1 is configured on a passive partition mapped on a storage node. Partition_1 and Partition_2 are active partitions. Note that under some circumstances, a partition, for instance Partition_2, can be duplicated. To be duplicated, Unit_2 and Unit_3 which are configured on Partition_2 have to provide only dynamically bound remote subprograms. Otherwise, a partition calling a remote subprogram on Unit_2 would not be able to statically determine where to perform the remote call between the two instances of Unit_2.



8.1.2 Categorization Pragmas

Library units can be categorized according to the role they play in a distributed program. A categorization pragma is a library unit pragma that restricts the kinds of declarations that can appear in a library unit and possibly in its child units, as well as the legal semantic dependences that the categorized unit can have. There are several categorization pragmas:

- Remote_Call_Interface
- Remote_Types
- Shared_Passive
- Pure

The following paragraphs do not present the detailed semantics of these pragmas (formal details will be found in the Ada Reference Manual). Their purpose is to give the reader an intuitive overview of the purpose of these pragmas. If a library unit is not categorized, this unit is called a normal unit and plays no special role in the distributed application. Such a unit is duplicated on any partition in which it is mentioned.

A parenthetical remark: to avoid the need for specific run-time libraries for the DSA, the notion of remote rendezvous does not exist in Ada: tasks cannot be invoked directly from one partition to another. Therefore, declarations of task types and general protected types with entries are not allowed in categorized Ada library units.

8.1.3 Pragma Declared Pure

This pragma is not specific to the Distributed Systems Annex. A pure package can appear in the context of any package, categorized or not. A pure package is a preelaborable package that does not contain variable data. It is particularly useful to define types, constants and subprograms shared by several categorized packages. In contrast, normal packages cannot appear in the context of categorized package declarations. Because a pure package has no state, it can be duplicated on several partitions.

8.1.4 Pragma Remote_Call_Interface

8.1.4.1 Overview of Pragma Remote_Call_Interface

Library units categorized with this pragma declare subprograms that can be called and executed remotely. An RCI unit acts as a server for remote calls. There is no memory space shared between server and clients. A subprogram call that invokes one such subprogram is a classical RPC operation; it is a statically bound operation, because the compiler can determine the identity of the subprogram being called.

Dynamically bound calls are provided through two mechanisms:

- The dereference of an access-to-subprogram value, i.e. a value whose type is a remote access-to-subprogram (RAS).
- A dispatching call whose controlling argument is an access-to-class-wide operand. The formal is a remote access-to-class-wide (RACW) type. These remote access types can be declared in RCI packages as well.

A remote access type (RAS or RACW) can be viewed as a fat pointer, that is to say a structure with a remote address and a local address (like an URL: `<protocol>`://-`<remote-machine>`/`<local-directory>`). The remote address must denote the host of the partition on which the entity has been created; the local address describes the local memory address within the host.

It is very unlikely that RCI units can be duplicated in the distributed system. An implementation may allow separate copies of a RCI unit as long as it ensures that the copies present a consistent state to all clients. In the general case, preserving consistency is very costly. For this reason, the implementation may require a RCI unit to be unique in the distributed system.

8.1.4.2 Regular Remote Subprograms (RCI)

In the following example, a RCIBank offers several remote services: Balance, Transfer, Deposit and Withdraw. On the caller side, the bank client uses the stub files of unit RCIBank. On the receiver side, the bank receiver uses the skeleton files of unit RCIBank including the body of this package.

```
package Types is
   pragma Pure;

   type Customer_Type is new String;
   type Password_Type is new String;
end Types;


with Types; use Types;
package RCIBank is
   pragma Remote_Call_Interface;

   function Balance
     (Customer : in Customer_Type;
      Password : in Password_Type)
      return Integer;

   procedure Transfer
     (Payer    : in Customer_Type;
      Password : in Password_Type;
```

```
       Amount   : in Positive;
       Payee    : in Customer_Type);

   procedure Deposit
     (Customer : in Customer_Type;
      Amount   : in Positive);

   procedure Withdraw
     (Customer : in Customer_Type;
      Password : in Password_Type;
      Amount   : in out Positive);
end RCIBank;
```

```
with Types; use Types;
with RCIBank; use RCIBank;
procedure RCIClient is
   B : Integer;
   C : Customer_Type := "rich";
   P : Password_Type := "xxxx";
begin
   B := Balance (C, P);
end RCIClient;
```

8.1.4.3 Remote Access to Subprograms (RAS)

In the following example, several mirroring banks offer their services through the same database. Each bank registers a reference to each of its services with a central bank. A client of the central bank requests a service from one of the mirroring banks. To satisfy requests, the RCI unit RASBank defines Balance_Type, a remote access to subprogram. (Recall that an access type declared in a remote unit has to be either remote access to subprogram or remote access to class wide type).

Note that to obtain a remote access to subprogram, the subprogram that delivers the remote access must be remote itself. Therefore, MirrorBank is a RCI library unit.

```
with Types; use Types;
package RASBank is
   pragma Remote_Call_Interface;

   type Balance_Type is access function
     (Customer : in Customer_Type;
      Password : in Password_Type)
      return Integer;

   procedure Register
      (Balance : in Balance_Type);

   function Get_Balance
      return Balance_Type;

   --  [...] Other services
end RASBank;
```

In the code below, a mirroring bank registers its services to the central bank.

```
with Types; use Types;
package MirrorBank is
   pragma Remote_Call_Interface;

   function Balance
     (Customer : in Customer_Type;
      Password : in Password_Type)
      return Integer;

   --  [...] Other services
end MirrorBank;
```

```
with RASBank, Types; use RASBank, Types;
package body MirrorBank is

   function Balance
     (Customer : in Customer_Type;
      Password : in Password_Type)
      return Integer is
   begin
      return Something;
   end Balance;

begin
   --  Register a dynamically bound remote subprogram (Balance)
   --  through a statically bound remote subprogram (Register)
   Register (Balance'Access);
   --  [...] Register other services
end MirrorBank;
```

In the code below, a central bank client asks for a mirroring bank and calls the Balance service of this bank by dereferencing a remote access type.

```
with Types; use Types;
with RASBank; use RASBank;
procedure BankClient is
   B : Integer;
   C : Customer_Type := "rich";
   P : Password_Type := "xxxx";
begin
   --  Through a statically bound remote subprogram (Get_Balance), get
   --  a dynamically bound remote subprogram. Dereference it to
   --  perform a dynamic invocation.
   B := Get_Balance.all (C, P);
end BankClient;
```

8.1.4.4 Remote Access to Class Wide Types (RACW)

A bank client is now connected to a bank through a terminal. The bank wants to notify a connected client, by means of a message on its terminal, when another client transfers a given amount of money to its account. In the following example, a terminal is designed as a distributed object. Each bank client will register its terminal object to the bank server

for further use. In the code below, Term_Type is the root type of the distributed terminal hierarchy.

```
with Types; use Types;
package Terminal is
   pragma Pure;

   type Term_Type is abstract tagged limited private;

   procedure Notify
     (MyTerm   : access Term_Type;
      Payer    : in Customer_Type;
      Amount   : in Integer) is abstract;

private
   --  implementation removed
end Terminal;
```

In the code below, the RCI unit RACWBank defines Term_Access, a remote access to class wide type. Term_Access becomes a reference to a distributed object. In the next section, we will see how to derive and extend Term_Type, how to create a distributed object and how to use a reference to it.

```
with Terminal, Types; use Terminal, Types;
package RACWBank is
   pragma Remote_Call_Interface;

   type Term_Access is access all Term_Type'Class;

   procedure Register
     (MyTerm   : in Term_Access;
      Customer : in Customer_Type;
      Password : in Password_Type);

   --  [...] Other services
end RACWBank;
```

8.1.4.5 Summary of Pragma Remote_Call_Interface

Remote call interface units:

- Allow subprograms to be called and executed remotely
- Allow statically bound remote calls (remote subprogram)
- Allow dynamically bound remote calls (remote access types)
- Forbid variables and non-remote access types
- Prevent specification from depending on normal units

8.1.5 Pragma Remote_Types

8.1.5.1 Overview of Pragma Remote_Types

Unlike RCI units, library units categorized with this pragma can define distributed objects and remote methods on them. Both RCI and RT units can define a remote access type as

described above (RACW). A subprogram defined in a RT unit is not a remote subprogram. Unlike RCI units, a RT unit can be duplicated on several partitions, in which case all its entities are distinct. This unit is duplicated on each partition in which it is defined.

8.1.5.2 Distributed Object

If we want to implement the notification feature proposed in the previous section, we have to derive Term_Type. Such a derivation is possible in a remote types unit like NewTerminal (see below). Any object of type New_Term_Type becomes a distributed object and any reference to such an object becomes a fat pointer or a reference to a distributed object (see Term_Access declaration in Section 8.1.4.4 [Remote Access to Class Wide Types (RACW)], page 58).

```
with Types, Terminal; use Types, Terminal;
package NewTerminal is
   pragma Remote_Types;

   type New_Term_Type is
      new Term_Type with null record;

   procedure Notify
     (MyTerm   : access New_Term_Type;
      Payer    : in Customer_Type;
      Amount   : in Integer);

   function Current return Term_Access;
end NewTerminal;
```

In the code below, a client registers his name and his terminal with RACWBank. Therefore, when any payer transfers some money to him, RACWBank is able to notify the client of the transfer of funds.

```
with NewTerminal, RACWBank, Types; use NewTerminal, RACWBank, Types;
procedure Term1Client is
   MyTerm   : Term_Access   := Current;
   Customer : Customer_Type := "poor";
   Password : Password_Type := "yyyy";
begin
   Register (MyTerm, Customer, Password);
   --  [...] Execute other things
end Term1Client;
```

In the code below, a second client, the payer, registers his terminal to the bank and executes a transfer to the first client.

```
with NewTerminal, RACWBank, Types; use NewTerminal, RACWBank, Types;
procedure Term2Client is
   MyTerm   : Term_Access   := Current;
   Payer    : Customer_Type := "rich";
   Password : Password_Type := "xxxx";
   Payee    : Customer_Type := "poor";
begin
   Register (MyTerm, Payer, Password);
   Transfer (Payer, Password, 100, Payee);
```

```
end Term2Client;
```

In the code below, we describe the general design of Transfer. Classical operations of Withdraw and Deposit are performed. Then, RACWBank retrieves the terminal of the payee (if present) and invokes a dispatching operation by dereferencing a distributed object Term. The reference is examined at run-time, and the execution of this operation takes place on the partition on which the distributed object resides.

```
with Types; use Types;
package body RACWBank is
   procedure Register
     (MyTerm   : in Term_Access;
      Customer : in Customer_Type;
      Password : in Password_Type) is
   begin
      Insert_In_Local_Table (MyTerm, Customer);
   end Register;

   procedure Transfer
     (Payer    : in Customer_Type;
      Password : in Password_Type;
      Amount   : in Positive;
      Payee    : in Customer_Type)
   is
      --  Find Customer terminal.
      Term : Term_Access
        := Find_In_Local_Table (Payee);
   begin
      Withdraw (Payer, Amount);
      Deposit  (Payee, Amount);
      if Term /= null then
         --  Notify on Payee terminal.
         Notify (Term, Payer, Amount);
      end if;
   end Transfer;

   --  [...] Other services
end RACWBank;
```

8.1.5.3 Transmitting Dynamic Structure

```
with Ada.Streams; use Ada.Streams;
package StringArrayStream is
   pragma Remote_Types;

   type List is private;
   procedure Append (L : access List; O : in String);
   function  Delete (L : access List) return String;

private
   --  implementation removed
end StringArrayStream;
```

Non-remote access types cannot be declared in the public part of a remote types unit. However, it is possible to define private non-remote access types as long as the user provides

its marshalling procedures, that is to say the mechanism needed to place a value of the type into a communication stream. The code below describes how to transmit a linked structure.

The package declaration provides a type definition of single-linked lists of unbounded strings. An implementation of the marshalling operations could be the following:

```
package body StringArrayStream is
   procedure Read
     (S : access Root_Stream_Type'Class;
      L : out List) is
   begin
      if Boolean'Input (S) then
         L := new Node;
         L.Content := new String'(String'Input (S));
         List'Read (S, L.Next);
      else
         L := null;
      end if;
   end Read;

   procedure Write
     (S : access Root_Stream_Type'Class;
      L : in List) is
   begin
      if L = null then
         Boolean'Output (S, False);
      else
         Boolean'Output (S, True);
         String'Output (S, L.Content.all);
         List'Write (S, L.Next);
      end if;
   end Write;

   --  [...] Other services
end StringArrayStream;
```

8.1.5.4 Summary of Remote Types Units

Remote types units:

- Support the definition of distributed objects
- Allow dynamically bound remote calls (via remote access types)
- Allow non-remote access types (with marshalling subprograms)
- Cannot have a specification that depends on normal units

8.1.6 Pragma Shared_Passive

8.1.6.1 Overview of Pragma Shared_Passive

The entities declared in such a categorized library unit are intended to be mapped on a virtual shared address space (file, memory, database). When two partitions use such a library unit, they can communicate by reading or writing the same variable in the shared unit. This supports the conventional shared variables paradigm. Entryless protected objects can be declared in these units, to provide an atomic access to shared data, thus implementing

a simple transaction mechanism. When the address space is a file or a database, the user can take advantage of the persistency features provided by these storage nodes.

8.1.6.2 Shared and Protected Objects

In the code below, we define two kinds of shared objects. External_Synchronization requires that the different partitions updating this data synchronize to avoid conflicting operations on shared objects. Internal_Synchronization provides a way to get an atomic operation on shared objects. Note that only entryless protected types are allowed in a shared passive unit; synchronization must be done with protected procedures.

```
package SharedObjects is
   pragma Shared_Passive;

   Max : Positive := 10;
   type Index_Type is range 1 .. Max;
   type Rate_Type is new Float;

   type Rates_Type is array (Index_Type) of Rate_Type;

   External_Synchronization : Rates_Type;

   protected Internal_Synchronization is
      procedure Set
        (Index : in Index_Type;
         Rate  : in Rate_Type);

      procedure Get
        (Index : in Index_Type;
         Rate  : out Rate_Type);
   private
   --  implementation removed
end SharedObjects;
```

8.1.6.3 Summary of Pragma Shared_Passive

Shared passive units:

- Allow direct access to data from different partitions
- Provide support for shared (distributed) memory
- Support memory protection by means of entryless protected objects
- Prevent specification from depending on normal units

8.1.7 More About Categorization Pragmas

8.1.7.1 Variables and Non-Remote Access Types

In RT or RCI package declarations, variable declarations are forbidden, and non-remote access types are allowed as long as their marshalling subprograms are explicitly provided (see Section 8.1.5.3 [Transmitting Dynamic Structure], page 61).

8.1.7.2 RPC Failures

Calls are executed at most once: they are made exactly one time or they fail with an exception. When a communication error occurs, *System.RPC.Communication_Error* is raised.

8.1.7.3 Exceptions

Any exception raised in a remote method or subprogram call is propagated back to the caller. Exception semantics are preserved in the regular Ada way.

```
package Internal is
   Exc : exception;
end Internal;


package RemPkg2 is
   pragma Remote_Call_Interface;

   procedure Subprogram;
end RemPkg2;


package RemPkg1 is
   pragma Remote_Call_Interface;

   procedure Subprogram;
end RemPkg1;
```

Let us say that RemPkg2, Internal and RemExcMain packages are on the same partition Partition_1 and that RemPkg1 is on partition Partition_2.

```
with RemPkg1, Ada.Exceptions; use Ada.Exceptions;
package body RemPkg2 is
   procedure Subprogram is
   begin
      RemPkg1.Subprogram;
   exception when E : others =>
      Raise_Exception (Exception_Identity (E), Exception_Message (E));
   end Subprogram;
end RemPkg2;


with Internal, Ada.Exceptions; use Ada.Exceptions;
package body RemPkg1 is
   procedure Subprogram is
   begin
      Raise_Exception (Internal.Exc'Identity, "Message");
   end Subprogram;
end RemPkg1;


with Ada.Text_IO, Ada.Exceptions; use Ada.Text_IO, Ada.Exceptions;
with RemPkg2, Internal;
procedure RemExcMain is
begin
```

```
   RemPkg2.Subprogram;
exception when E : Internal.Exc =>
   Put_Line (Exception_Message (E)); -- Output "Message"
end RemExcMain;
```

When RemPkg1.Subprogram on Partition_1 raises Internal.Exc, this exception is propagated back to Partition_2. As Internal.Exc is not defined on Partition_2, it is not possible to catch this exception without an exception handler **when others**. When this exception is reraised in RemPkg1.Subprogram, it is propagated back to Partition_1. But this time, Internal.Exc is visible and can be handled as we would in a single-partition Ada program. Of course, the exception message is also preserved.

8.1.7.4 Pragma Asynchronous

By default, a remote call is blocking: the caller waits until the remote call is complete and the output stream is received. Just like a normal (nonremote) call, the caller does not proceed until the call returns. By contrast, a remote subprogram labeled with pragma Asynchronous allows statically and dynamically bound remote calls to it to be executed asynchronously. A call to an asynchronous procedure doesn't wait for the completion of the remote call, and lets the caller continue its execution. The remote procedure must have only **in** parameters, and any exception raised during the execution of the remote procedure is lost.

When pragma Asynchronous applies to a regular subprogram with **in** parameters, any call to this subprogram will be executed asynchronously. The following declaration of AsynchronousRCI.Asynchronous gives an example.

```
package AsynchronousRCI is
   pragma Remote_Call_Interface;

   procedure Asynchronous (X : Integer);
   pragma Asynchronous (Asynchronous);

   procedure Synchronous  (X : Integer);

   type AsynchronousRAS is access procedure (X : Integer);
   pragma Asynchronous (AsynchronousRAS);
end AsynchronousRCI;


package AsynchronousRT is
   pragma Remote_Types;

   type Object is tagged limited private;

   type AsynchronousRACW is access all Object'Class;
   pragma Asynchronous (AsynchronousRACW);

   procedure Asynchronous (X : Object);
   procedure Synchronous  (X : in out Object);
   function Create return AsynchronousRACW;

private
   --  implementation removed
```

```
end AsynchronousRT;
```

A pragma Asynchronous may apply to a remote access-to-subprogram (RAS) type. An asynchronous RAS can be both asynchronous and synchronous depending on the designated subprogram. For instance, in the code below, remote call (1) is asynchronous but remote call (2) is synchronous.

A pragma Asynchronous may apply to a RACW as well. In this case, the invocation of **any** method with **in** parameters is *always* performed asynchronously. Remote method invocation (3) is asynchronous but remote method invocation (4) is synchronous.

```
with AsynchronousRCI, AsynchronousRT;
use AsynchronousRCI, AsynchronousRT;
procedure AsynchronousMain is
   RAS  : AsynchronousRAS;
   RACW : AsynchronousRACW := Create;
begin
   --  Asynchronous Dynamically Bound Remote Call (1)
   RAS := AsynchronousRCI.Asynchronous'Access;
   RAS (0);  --  Abbrev for RAS.all (0)
   --  Synchronous Dynamically Bound Remote Call (2)
   RAS := AsynchronousRCI.Synchronous'Access;
   RAS (0);
   --  Asynchronous Dynamically Bound Remote Call (3)
   Asynchronous (RACW.all);
   --  Synchronous Dynamically Bound Remote Call (4)
   Synchronous (RACW.all);
end AsynchronousMain;
```

This feature supports the conventional message passing paradigm. The user must be aware that this paradigm, and asynchronous remote calls in particular, has several drawbacks:

- It violates the normal semantics of calls; the caller proceeds without awaiting the return. The semantics are more similar to a "remote goto" than a remote call
- It prevents easy development and debugging in a non-distributed context
- It can introduce race conditions

To illustrate the latter, let us take the following example:

```
package Node2 is
   pragma Remote_Call_Interface;

   procedure Send (X : Integer);
   pragma Asynchronous (Send);
end Node2;


package body Node2 is
   V : Integer := 0;
   procedure Send (X : Integer) is
   begin
      V := X;
   end Send;
end Node2;
```

```
package Node1 is
   pragma Remote_Call_Interface;

   procedure Send (X : Integer);
   pragma Asynchronous (Send);
end Node1;
```

```
with Node2;
package body Node1 is
   procedure Send (X : Integer) is
   begin
      Node2.Send (X);
   end Send;
end Node1;
```

```
with Node1, Node2;
procedure NonDeterministic is
begin
   Node1.Send (1);
   Node2.Send (2);
end NonDeterministic;
```

Let us say that Main is configured on Partition_0, Node1 on Partition_1 and Node2 on Partition_2. If Node1.Send and Node2.Send procedures were synchronous or if no latency was introduced during network communication, we would have the following RPC order: Main remotely calls Node1.Send which remotely calls Node2.Send which sets V to 1. Then, Main remotely calls Node2.Send and sets V to 2.

Now, let us assume that both Send procedures are asynchronous and that the connection between Partition_1 and Partition_2 is very slow. The following scenario can very well occur. Main remotely calls Node1.Send and is unblocked. Immediately after this call, Main remotely calls Node2.Send and sets V to 2. Once this is done, the remote call to Node1.Send completes on Partition_1 and it remotely calls Node2.Send which sets V to 1.

8.1.7.5 Pragma All_Calls_Remote

A pragma All_Calls_Remote in a RCI unit forces remote procedure calls to be routed through the communication subsystem even for a local call. This eases the debugging of an application in a non-distributed situation that is very close to the distributed one, because the communication subsystem (including marshalling and unmarshalling procedures) can be exercised on a single node.

In some circumstances, a non-distributed application can behave differently from an application distributed on only one partition. This can happen when both All_Calls_Remote and Asynchronous features are used at the same time (see Section 8.1.7.4 [Pragma Asynchronous], page 65 for an example). Another circumstance occurs when the marshalling operations raise an exception. In the following example, when unit ACRRCI is a All_Calls_-Remote package, the program raises Program_Error. When unit ACRRCI is no longer a All_Calls_Remote package, then the program completes silently.

```
with Ada.Streams; use Ada.Streams;
package ACRRT is
   pragma Remote_Types;
   type T is private;
private
   --  implementation removed
end ACRRT;
```

```
package body ACRRT is
   procedure Read
     (S : access Root_Stream_Type'Class;
      X : out T) is
   begin
      raise Program_Error;
   end Read;

   procedure Write
     (S : access Root_Stream_Type'Class;
      X : in T) is
   begin
      raise Program_Error;
   end Write;
end ACRRT;
```

```
with ACRRT; use ACRRT;
package ACRRCI is
   pragma Remote_Call_Interface;
   pragma All_Calls_Remote;

   procedure P (X : T);
end ACRRCI;
```

```
package body ACRRCI is
   procedure P (X : T) is
   begin
      null;
   end P;
end ACRRCI;
```

```
with ACRRCI, ACRRT;
procedure ACRMain is
   X : ACRRT.T;
begin
   ACRRCI.P (X);
end ACRMain;
```

8.1.7.6 Generic Categorized Units

```
generic
package GenericRCI is
   pragma Remote_Call_Interface;
```

```
   procedure P;
end GenericRCI;


with GenericRCI;
package RCIInstantiation is new GenericRCI;
pragma Remote_Call_Interface (RCIInstantiation);


with GenericRCI;
package NormalInstantiation is new GenericRCI;
```

Generic units may be categorized. Instances do not automatically inherit the categorization of their generic units, but they can be categorized explicitly. If they are not, instances are normal compilation units. Like any other categorized unit, a categorized instance must be at the library level, and the restrictions of categorized units apply on instantiation (in particular on generic formal parameters).

8.1.7.7 Categorization Unit Dependencies

Each categorization pragma has very specific visibility rules. As a general rule, RCI > RT > SP > Pure, where the comparison indicates allowed semantic dependencies. This means that a Remote_Types package can make visible in its specification only Remote_Types, Shared_Passive and Pure units.

8.2 Partition Communication Subsystem

8.2.1 Marshalling and Unmarshalling Operations

The Partition Communication Subsystem (PCS) is the runtime library for distributed features. It marshals and unmarshals client and server requests into a data stream suitable for network transmission.

Parameter streams are normally read and written using four attributes:

- Write: write an element into a stream, valid only for constrained types
- Read: read a constrained element from a stream
- Output: same as Write, but write discriminants or array bounds as well if needed
- Input: same as Read, but read discriminants or bounds from the stream (the Input attribute denotes a function)

An Ada compiler provides default 'Read and 'Write operations. But it is up to the implementation of the PCS to provide default 'Read and 'Write to ensure proper operation between heterogeneous architectures (see Section 8.5.7.2 [Heterogeneous System], page 91).

The user can override these operations, except for predefined types. Overriding with a custom version provides the user with a way to debug its application (even outside of the Distributed Systems Annex). On the other hand, remaining with the default implementation allows the user to take advantage of optimized and portable representations provided by the PCS.

```
with Ada.Streams; use Ada.Streams;
package New_Integers is
   pragma Pure;

   type New_Integer is new Integer;

   procedure Read
     (S : access Root_Stream_Type'Class;
      V : out New_Integer);
   procedure Write
     (S : access Root_Stream_Type'Class;
      V : in New_Integer);

   for New_Integer'Read  use Read;
   for New_Integer'Write use Write;
end New_Integers;


package body New_Integers is
   procedure Read
     (S : access Root_Stream_Type'Class;
      V : out New_Integer)
   is
      B : String := String'Input (S);
   begin
      V := New_Integer'Value (B);
   end Read;

   procedure Write
     (S : access Root_Stream_Type'Class;
      V : in New_Integer)
   is
   begin
      String'Output (S, New_Integer'Image (V));
   end Write;
end New_Integers;
```

The language forces the user to provide Read and Write operations for non-remote access types. Transmitting an access value by dumping its content into a stream makes no sense when the value is going to be transmitted to another partition (with a different memory space). To transmit non-remote access types see Section 8.1.5.3 [Transmitting Dynamic Structure], page 61.

8.2.2 Incorrect Remote Dispatching

When a remote subprogram takes a class wide argument, there is a risk of using an object of a derived type that will not be clean enough to be transmitted. For example, given a type called Root_Type, if a remote procedure takes a Root_Type'Class as an argument, the user can call it with an instance of Derived_Type that is Root_Type enriched with a field of a task type. This will lead to a non-communicable type to be transmitted between partitions.

To prevent this, paragraph E.4(18) of the Ada Reference Manual explains that any actual type used as parameter for a remote call whose formal type is a class wide type must be declared in the visible part of a Pure or Remote_Types package. This property also holds

for remote functions returning class wide types. To summarize, the actual type used should have been eligible for being declared where the root type has been declared. If a 'bad' object is given to a remote subprogram, *Program_Error* will be raised at the point of the call.

8.2.3 Partition Ids

U'Partition_ID identifies the partition where the unit U has been elaborated. For this purpose, the PCS provides an integer type Partition_ID to uniquely designate a partition. Note that a Partition_ID is represented as a universal integer, and has no meaning outside of the PCS. The RM requires that two partitions of a distributed program have different Partition_ID's at a given time. A Partition_ID may or may not be assigned statically (at compile or link time). A Partition_ID may or may not be related to the physical location of the partition.

Partition_ID's can be used to check whether a RCI package is configured locally.

```
with RCI;
with Ada.Text_IO;
procedure Check_PID is
begin
   if RCI'Partition_ID = Check_PID'Partition_ID then
      Ada.Text_IO.Put_Line ("package RCI is configured locally");
   else
      Ada.Text_IO.Put_Line ("package RCI is configured remotely");
   end if;
end Check_PID;
```

8.2.4 Concurrent Remote Calls

It is not defined by the PCS specification whether one or more threads of control should be available to process incoming messages and to wait for their completion. But the PCS implementation is required to be reentrant, thereby allowing concurrent calls on it to service concurrent remote subprogram calls into the server partition. This means that at the implementation level the PCS manages a pool of helper tasks. This (apart from performance) is invisible to the user.

8.2.5 Consistency and Elaboration

A library unit is consistent if the same version of its declaration is used in all units that reference it. This requirement applies as well to a unit that is referenced in several partitions of a distributed program. If a shared passive or RCI library unit U is included in some partition P, It is a bounded error to elaborate another partition P1 of a distributed program that that depends on a different version of U. As a result of this error, Program_Error can be raised in one or both partitions during elaboration.

U'Version yields a string that identifies the version of the unit declaration and any unit declaration on which it depends. U'Version_Body yields a string that identifies the version of the unit body. These attributes are used by the PCS to verify the consistency of an application.

After elaborating the library units, but prior to invoking the main subprogram, the PCS checks the RCI unit versions, and then accept any incoming RPC. To guarantee that it is

safe to call receiving stubs, any incoming RPC is kept pending until the partition completes its elaboration.

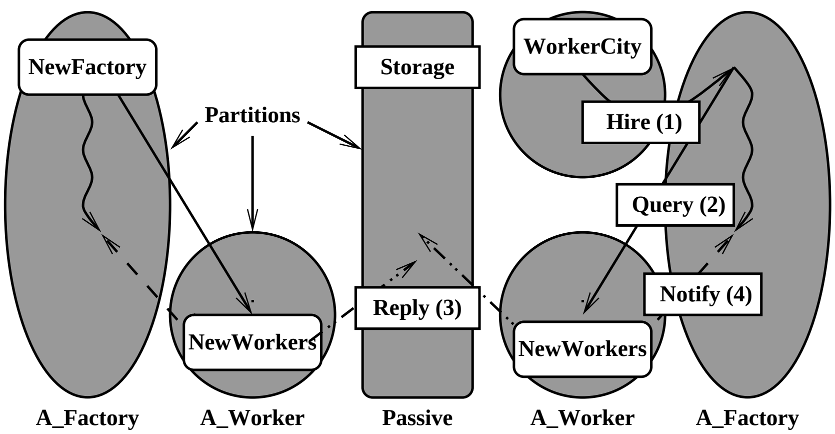
8.2.6 Abortion and Termination

If a construct containing a remote call is aborted, the remote subprogram call is cancelled. Whether the execution of the remote subprogram is immediately aborted as a result of the cancellation is implementation defined.

An active partition terminates when its environment task terminates. In other terms, a partition cannot terminate before the Ada program itself terminates. The standard termination mechanism applies, but can be extended with extra rules (see Section 8.5.4.22 [Partition Attribute Termination], page 85 for examples).

8.3 Most Features in One Example

The example shown on the following figure highlights most of the features of DSA. The system is based on a set of factories and workers and a storage. Each entity is a partition itself. A factory hires a worker from a pool of workers (hire - 1) and assigns a job (query - 2) to him. The worker performs the job and saves the result (reply - 3) in a storage common to all the factories. The worker notifies the factory of the end of his job (notify - 4).



When a worker has completed his job, the result must be saved in a common storage. To do this, we define a protected area in SP package Storage (see following code). An entryless protected object ensures atomic access to this area.

```
package Storage is
   pragma Shared_Passive;

   protected Queue is
      procedure Insert (Q, R : Integer);
      procedure Remove
        (Q : in Integer;
         R : out Integer);
   private
   --  implementation removed
```

```
end Storage;
```

Common is a Remote_Types package that defines most of the remote services of the above system (see following code). First, we define a way for the workers to signal the completion of his job. This callback mechanism is implemented using RAS Notify.

```
with Storage; use Storage;
package Common is
   pragma Remote_Types;

   type Notify is
      access procedure (Q : Integer);
   pragma Asynchronous (Notify);

   type Worker is
      abstract tagged limited private;
   procedure Assign
     (W : access Worker;
      Q : in Integer;
      N : in Notify) is abstract;

   type Any_Worker is
      access all Worker'Class;
   pragma Asynchronous (Any_Worker);

private
   --  implementation removed
end Common;
```

We define an abstract tagged type Worker which is intended to be the root type of the whole distributed objects hierarchy. Assign allows a factory to specify a job to a worker and a way for the worker to signal its employer the completion of this job. Any_Worker is a remote access to class wide type (RACW). In other words, it is a reference to a distributed object of any derived type from Worker class. Note that the two remote access types (Any_Worker and Notify) are declared as asynchronous. Therefore, any override of Assign will be executed asynchronously. To be asynchronous, an object of type Notify has to be a reference to an asynchronous procedure.

NewWorker is derived from type Worker and Assign is overridden.

```
with Common, Storage; use Common, Storage;
package NewWorkers is
   pragma Remote_Types;

   type NewWorker is new Worker with private;

   procedure Assign
     (W : access NewWorker;
      Q : Integer;
      N : Notify);
private
   --  implementation removed
end NewWorkers;
```

The following code shows how to derive a second generation of workers NewNewWorker from the first generation NewWorker. As mentioned above, this RT package can be duplicated on several partitions to produce several types of workers and also several remote workers.

```
with Common, Storage, NewWorkers; use Common, Storage, NewWorkers;
package NewNewWorkers is
   pragma Remote_Types;

   type NewNewWorker is new NewWorker with private;

   procedure Assign
     (W : access NewNewWorker;
      Q : Integer;
      N : Notify);
private
   --  implementation removed
end NewNewWorkers;
```

In the following code, we define a unique place where workers wait for jobs. WorkerCity is a Remote_Call_Interface package with services to hire and free workers. Unlike Remote_Types packages, Remote_Call_Interface packages cannot be duplicated, and are assigned to one specific partition.

```
with Common; use Common;
package WorkerCity is
   pragma Remote_Call_Interface;

   procedure Insert (W : in  Any_Worker);
   procedure Remove (W : out Any_Worker);
end WorkerCity;
```

In order to use even more DSA features, Factory is defined as a generic RCI package (see sample above). Any instantiation defines a new factory (see sample above). To be RCI, this instantiation has to be categorized once again.

```
with Storage; use Storage;
generic
package Factory is
   pragma Remote_Call_Interface;

   procedure Notify (Q : Integer);
   pragma Asynchronous (Notify);
end Factory;
```

```
with Factory;
package NewFactory is new Factory;
pragma Remote_Call_Interface (NewFactory);
```

8.4 A small example of a DSA application

In this section we will write a very simple client-server application using PolyORB DSA. The server will provide a `Remote Call Interface` composed of a single `Echo_String` function that will take a String and return it to the caller.

Here is the code for the server:

'`server.ads`':

```
package Server is
   pragma Remote_Call_Interface;
   function Echo_String (S : String) return String;
end Server;
```

'`server.adb`':

```
package body Server is

   function Echo_String (S : String) return String is
   begin
      return S;
   end Echo_String;

end Server;
```

And here is the code for the client:

'`client.adb`':

```
with Ada.Text_IO; use Ada.Text_IO;
with Server;
procedure Client is
begin
   Put_Line ("The client has started!");
   Put ("Thus spake my server upon me:");
   Put_Line (Server.Echo_String ("Hi!"));
end Client;
```

For more details about the Distributed Systems Annex, see the Ada Reference Manual *[ISO06]*.

8.5 Building a DSA application with PolyORB

This section describes how to build a complete distributed Ada application using the PolyORB implementation of the DSA.

8.5.1 Introduction to PolyORB/DSA

A distributed Ada application comprises a number of partitions which can be executed concurrently on the same machine or, and this is the interesting part, can be distributed on a network of machines. The way in which partitions communicate is described in Annex E of the Ada Reference Manual.

A partition is a set of compilation units that are linked together to produce an executable binary. A distributed program comprises two or more communicating partitions.

The Distributed Systems Annex (DSA) does not describe how a distributed application should be configured. It is up to the user to define what are the partitions in his program and on which machines they should be executed.

The tool `po_gnatdist` and its configuration language allows the user to partition his program and to specify the machines on which the individual partitions are to execute.

`po_gnatdist` reads a configuration file (whose syntax is described in section Section 8.5.4 [The Configuration Language], page 76) and builds several executables, one for each partition. It also takes care of launching the different partitions (default) with parameters that can be specific to each partition.

8.5.2 How to Configure a Distributed Application

- Write a non-distributed Ada application, to get familiar with the PolyORB environment. Use the categorization pragmas to specify the packages that can be called remotely.
- When this non-distributed application is working, write a configuration file that maps the user categorized packages onto specific partitions. This concerns particularly remote call interface and remote types packages. Specify the main procedure of the distributed application (see Section 8.5.4.6 [Partition Attribute Main], page 79).
- Type 'po_gnatdist <*configuration-file*>'.
- Start the distributed application by invoking the start-up shell script or default Ada program (depending on the Starter option, see Section 8.5.4.7 [Pragma Starter], page 80).

8.5.3 Gnatdist Command Line Options

```
po_gnatdist [switches] configuration-file [list-of-partitions]
```

The switches of `po_gnatdist` are, for the time being, exactly the same as those of gnatmake, with the addition of `--PCS`, which allows the user to override the default selection of distribution runtime library (PCS). By default `po_gnatdist` outputs a configuration report and the actions performed. The switch -n allows `po_gnatdist` to skip the first stage of recompilation of the non-distributed application.

The names of all configuration files must have the suffix `.cfg`. There may be several configuration files for the same distributed application, as the user may want to use different distributed configurations depending on load and other characteristics of the computing environment.

If a list of partitions is provided on the command line of the po_gnatdist command, only these partitions will be built. In the following configuration example, the user can type :

```
po_gnatdist <configuration> <partition_2> <partition_3>
```

8.5.4 The Configuration Language

The configuration language is *Ada-like*. As the capabilities of PolyORB will evolve, so will this configuration language. Most of the attributes and pragmas can be overridden at run-time by command line arguments or environment variables.

8.5.4.1 Language Keywords

All the Ada keywords are reserved keywords of the configuration language. `po_gnatdist` generates full Ada code in order to build the different executables. To avoid naming conflicts between Ada and the configuration language, all the Ada keywords have been reserved even if they are not used in the configuration language.

There are three new keywords:

- *configuration* to encapsulate a configuration
- *Partition* that is a predefined type to declare partitions

8.5.4.2 Pragmas and Representation Clauses

It is possible to modify the default behavior of the configuration via a pragma definition.

```
PRAGMA ::=
   pragma PRAGMA_NAME [(PRAGMA_ARGUMENTS)];
```

It is also possible to modify the default behavior of all the partitions via an attribute definition clause applied to the predefined type **Partition**.

```
REPRESENTATION_CLAUSE ::=
   for Partition'ATTRIBUTE_NAME use ATTRIBUTE_ARGUMENTS;
```

It is also possible to modify the default behavior of a given partition via an attribute definition clause applied to the partition itself.

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'ATTRIBUTE_NAME use ATTRIBUTE_ARGUMENTS;
```

When an attribute definition clause is applied to a given object of a predefined type, this overrides any attribute definition of the predefined type. In the next sections, attributes apply to a given object rather than to the predefined type.

8.5.4.3 Configuration Declaration

The distribution of one or several Ada programs is described by a single configuration unit. This configuration unit has a specification part and an optional body part. A configuration unit is declared as an Ada procedure would be. The keyword **configuration** is reserved for this purpose.

```
CONFIGURATION_UNIT ::=
   configuration IDENTIFIER is
      DECLARATIVE_PART
   [begin
      SEQUENCE_OF_STATEMENTS]
   end [IDENTIFIER];
```

8.5.4.4 Partition Declaration

In the declarative part, the user declares his partitions and can change their default behavior. `po_gnatdist` provides a predefined type **Partition**. The user can declare a list of partitions and can also initialize these partitions with an initial list of Ada units.

```
DECLARATIVE_PART ::= {DECLARATIVE_ITEM}

DECLARATIVE_ITEM ::=
   PARTITION_DECLARATION
 | REPRESENTATION_CLAUSE
 | SUBPROGRAM_DECLARATION
 | PRAGMA

SUBPROGRAM_DECLARATION ::=
   MAIN_PROCEDURE_DECLARATION
 | PROCEDURE_DECLARATION
 | FUNCTION_DECLARATION

PARTITION_DECLARATION ::=
   DEFINING_IDENTIFIER_LIST : Partition
      [:= ENUMERATION_OF_ADA_UNITS];

DEFINING_IDENTIFIER_LIST ::=
   DEFINING_IDENTIFIER {, DEFINING_IDENTIFIER}

STATEMENT ::=
   IDENTIFIER := ENUMERATION_OF_ADA_UNITS;

SEQUENCE_OF_STATEMENTS ::=
   STATEMENT {STATEMENT}
```

Once declared, a partition is an empty list of Ada units. The operator `":="` adds the Ada units list on the right side to the current list of Ada units that are already mapped to the partition. This is a non-destructive operation. Whether a unit is a relevant Ada unit or not is checked later on by the back-end of `po_gnatdist`. These assignments can occur in the declarative part as well as in the body part.

```
ENUMERATION_OF_ADA_UNITS ::= ({ADA_UNIT {, ADA_UNIT}});
```

8.5.4.5 Location Declaration

There are several kinds of location in the configuration language. We shall present them in the next subsections, but here is a short overview of these locations:

- Boot_Location defines the network locations to use to communicate with the the boot server during the boot phase
- Self_Location defines the network locations to use by others to communicate with the current partition
- Data_Location defines the data storage location used by the current partition to map its shared passive units

A location is composed of a support name and a specific data for this support. For instance, a network location is composed of a protocol name like *tcp* and a protocol data like <*machine*>:<*port*>. A storage location is composed of a storage support name like *dfs* (for Distributed File System) and a storage support data like a directory */dfs/glade*.

```
LOCATION      ::= ([Support_Name =>] STRING_LITERAL,
```

```
                        [Support_Data =>] STRING_LITERAL)

LOCATION_LIST ::= (LOCATION [,LOCATION)])
```

Note that a location may have an undefined or incomplete support data. In this case, the support is free to compute a support data. For instance, ("tcp", "") specifies that the protocol is used but that the protocol data <*machine*>:<*port*> is to be determined by the protocol itself.

A location or a list of locations can be can be concatenated into a single string to be used as a command line option or an environment variable (see Section 8.5.5 [Partition Runtime Parameters], page 90).

If a partition wants to communicate with another partition once the location list of the latter is known, the caller will use the first location of the callee whose protocol is locally available. For instance, if a callee exports three locations ("N1", "D1"), ("N2", "D2") and ("N3", "D3"), a caller with protocols N2 and N3 locally available will try to communicate with the callee using the protocol of name N2 and of specific data D2.

8.5.4.6 Partition Attribute Main

Basically, the distributed system annex (DSA) helps the user in building a distributed application from a non-distributed application (Of course, this is not the only possible model offered by DSA). The user can design, implement and test his application in a non-distributed environment, and then should be able to switch from the non-distributed case to a distributed case. As mentioned before, this two-phase design approach has several advantages.

In a non-distributed case, the user executes only one main executable possibly with a name corresponding to the main unit name of his application. With `po_gnatdist`, in a distributed case, a main executable with a name corresponding to the main unit name is responsible for starting the entire distributed application. Therefore, the user can start his application the same way he used to do in the non-distributed case.

For this reason, the configuration language provides a way to declare the main procedure of the non-distributed application.

```
MAIN_PROCEDURE_IDENTIFIER ::=
  ADA_UNIT
MAIN_PROCEDURE_DECLARATION ::=
   procedure MAIN_PROCEDURE_IDENTIFIER is in PARTITION_IDENTIFIER;
```

In this case, the partition in which the main procedure has been mapped is called the main partition. It includes in its code a call to this main procedure. The main partition has an additional specific role, because the boot server is located on it (see Section 8.5.7 [PolyORB PCS Internals], page 91).

The main procedures for the other partitions have a null body. However, the user can also modify this behavior by providing an alternate main procedure. To do this, an alternate main subprogram has to be declared and assigned to the partition Main attribute.

```
PROCEDURE_DECLARATION ::=
   procedure PROCEDURE_IDENTIFIER;
```

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Main use PROCEDURE_IDENTIFIER;
```

8.5.4.7 Pragma Starter

As a default, the main executable is a full Ada starter procedure. That means that it launches all the other partitions from an Ada program. The pragma Starter allows the user to ask for one starter or another. When the partition host is not statically defined (see Section 8.5.4.17 [Partition Attribute Host], page 83), the starter subprogram will ask for it interactively when it is executed.

```
CONVENTION_LITERAL ::= Ada   |
                       Shell |
                       None

PRAGMA ::=
   pragma Starter ([Convention =>] CONVENTION_LITERAL);
```

- The default method consists in launching partitions from the main partition Ada subprogram using a remote shell (see below).
- The user may ask for a Shell script that starts the different partitions one at a time on the appropriate remote machines, using a remote shell. As the Ada starter, the Shell script starter ask for partition hosts interactively when a partition host is not already defined. Having a textual shell script allows the user to edit it and to modify it easily.
- The user may ask for a None starter. In this case, it is up to the user to launch the different partitions.

8.5.4.8 Pragma Remote_Shell

When pragma Starter is Ada or Shell, the main partition launches the other partitions. The remote shell used as a default is determined during PolyORB configuration and installation. It is either rsh, remsh or the argument passed to –with-rshcmd=[ARG]. The pragma Remote_Shell allows the user to override the default.

```
PRAGMA ::=
   pragma Remote_Shell
     ([Command =>] STRING_LITERAL,
      [Options =>] STRING_LITERAL);
```

The Command parameter indicates the name of the remote shell command name and the Options parameter corresponds to the additional flags to pass to the remote shell command.

8.5.4.9 Pragma Name_Server

```
NAME_SERVER_LITERAL ::= Embedded   |
                       Standalone

PRAGMA ::=
   pragma Name_Server ([Name_Server_Kind =>] NAME_SERVER_LITERAL);
```

By default, partitions in a PolyORB/DSA application rely on an external, stand-alone name server launched by the user, and whose location is retrieved from runtime configuration.

A pragma Name_Server with parameter Embedded can be used to request the PCS to instead set up a name server within the main partition. If the Ada starter is used, the location of the name server is passed automatically to slave partitions.

8.5.4.10 Pragma Boot_Location

When a partition starts executing, one of the first steps consists in a connection to the boot server. This pragma provides one or more locations in order to get a connection with the boot server.

```
PRAGMA ::=
   PRAGMA_WITH_NAME_AND_DATA
 | PRAGMA_WITH_LOCATION
 | PRAGMA_WITH_LOCATION_LIST

PRAGMA_WITH_NAME_AND_DATA ::=
   pragma Boot_Location
     ([Protocol_Name =>] STRING_LITERAL,
      [Protocol_Data =>] STRING_LITERAL);

PRAGMA_WITH_LOCATION ::=
   pragma Boot_Location ([Location =>] LOCATION);

PRAGMA_WITH_LOCATION_LIST ::=
   pragma Boot_Location ([Locations =>] LOCATION_LIST);
```

This boot server location can be concatenated into a single string to be used as a command line option or an environment variable (see Section 8.5.5 [Partition Runtime Parameters], page 90).

Note: pragma Boot_Server is now obsolete. It is recommended to use pragma Boot_Location. This wording is more consistent with the rest of the configuration language (see Self_Location ⟨undefined⟩ [Partition Option self_location], page ⟨undefined⟩ and Data_Location ⟨undefined⟩ [Partition Option data_location], page ⟨undefined⟩).

8.5.4.11 Partition Attribute Self_Location

Except for the boot partition on which the boot server is located, a partition is reachable through a dynamically computed location (for instance, the partition looks for a free port when the protocol is tcp). The user may want such a partition to be reachable from a given location, especially if the user wants to make this partition a boot mirror. To do so, he can force the partition location with self_location feature.

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Self_Location use LOCATION;
 | for PARTITION_IDENTIFIER'Self_Location use LOCATION_LIST;
```

If the attribute definition clause applies to the predefined type **Partition**, the locations have to be incomplete. Otherwise, all the partitions would be reachable through the same locations, which is definitively not recommended.

When an attribute self_location definition clause applies to a given partition, the protocol units needed for this partition are linked in the executable. By default, when the self_location attribute is not redefined, the default protocol used by the partition and loaded in its executable is the *tcp* protocol.

8.5.4.12 Partition Attribute Passive

By default, a partition is an active partition. This attribute allows to define a passive partition. In this case, `po_gnatdist` checks that only shared passive units are mapped on the partition. As this partition cannot register itself, its location is hard-coded in all the partitions that depend on its shared passive units.

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Passive use BOOLEAN_LITERAL;
```

8.5.4.13 Partition Attribute Data_Location

Shared passive units can be mapped on passive or active partitions. In both cases, it is possible to choose the data storage support and to configure it with the specific data of a location.

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Data_Location use LOCATION;
 | for PARTITION_IDENTIFIER'Data_Location use LOCATION_LIST;
```

When an attribute data_location definition clause applies to a given partition, the data storage support units needed for this partition are linked in the executable. By default, when the data_location attribute is not redefined, the default storage support used by the partition and loaded in its executable is the *dfs* support. *dfs*, Distributed File System, is a storage support available as soon as files can be shared between partitions.

It is not possible to map the different shared passive units of a given partition on different data storage locations. PolyORB requires all the shared passive units of a given partition to be mapped on the same storage support. When the attribute data_location applied to a partition is a list of locations, all the storage support units needed for this partition are linked in the executable. By default, only the first one is activated. The user can choose to change the activated support by another one specified in the location list. This can be done using the partition option data_location (see ⟨undefined⟩ [Partition Option data_location], page ⟨undefined⟩).

As passive partitions cannot be activated, it is not possible to provide a location list as a data_location attribute. It is not possible to change dynamically its location either.

8.5.4.14 Partition Attribute Allow_Light_PCS

On some circumstances, `po_gnatdist` can detect that a partition does not need the full PCS functionalities. This occurs in particular when the partition does use any task, any RCI unit or any RACW object. Therefore, the partition does not receive any message that is not a reply to a previous request. In this case, the PCS does not drag in the tasking library and a light PCS is linked in the partition executable. This specific configuration is automatically determined by `po_gnatdist` with the ALI file information.

This optimization can be inappropriate especially when the user wants to use the "Distributed Shared Memory" storage support which runs Li and Hudak's algorithm. In this case, messages are exchanged without being replies to previously sent requests and the normal PCS should be linked instead of the light one. Note also that `po_gnatdist` cannot know for sure that the DSM storage support assigned at configuration time is used at run-time. The user can configure this optimization with the following attribute.

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Allow_Light_PCS use BOOLEAN_LITERAL;
```

8.5.4.15 Pragma Priority

It might be necessary for real-time applications to get control over the priority at which a remote procedure call is executed. By default, the PCS sends the priority of the client to the server which sets the priority of an anonymous task to this value. The pragma Priority allows to decide which priority policy should apply in the distributed application.

```
PRIORITY_POLICY_LITERAL ::= Server_Declared
                          | Client_Propagated

PRAGMA ::=
   pragma Priority ([Policy =>] PRIORITY_POLICY_LITERAL);
```

- The default policy Client_Propagated consists in propagating the client priority to the server.
- The policy Server_Declared consists in executing the remote procedure call at a priority specific to the partition. This priority can be set using the partition attribute Priority.

8.5.4.16 Partition Attribute Priority

This attribute allows to set the priority at which level a remote procedure call is executed on a server when the priority policy is Server_Declared. By default, the default priority of the anonymous task is the default task priority.

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Priority use INTEGER_LITERAL;
```

8.5.4.17 Partition Attribute Host

Logical nodes (or partitions) can be mapped onto physical nodes. The host-name can be either a static or dynamic value. In case of a static value, the expression is a string literal. In case of a dynamic value, the representation clause argument is a function that accepts a string as parameter and that returns a string value. When the function is called, the partition name is passed as parameter and the host-name is returned.

```
FUNCTION_DECLARATION ::=
   function FUNCTION_IDENTIFIER
     (PARAMETER_IDENTIFIER : [in] String)
      return String;
```

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Host use STRING_LITERAL;
  | for PARTITION_IDENTIFIER'Host use FUNCTION_IDENTIFIER;
```

The signature of the function must be the following : it takes a string parameter which corresponds to a partition name. It returns a string parameter which corresponds to the host-name. The function that returns the host-name can be an Ada function (default) or a shell script. A pragma Import is used to import a function defined in Ada or in Shell (see Section 8.5.4.18 [Pragma Import], page 84).

This function is called on the main partition by the PCS to launch a given partition on a given logical node. In case of load balancing, the function can return the most appropriate among a set of hosts.

8.5.4.18 Pragma Import

Two kinds of subprograms are allowed in the configuration language. A main procedure is used as a partition Main attribute and a function is used as a partition Host attribute.

```
PROCEDURE_DECLARATION ::=
     procedure PROCEDURE_IDENTIFIER;
FUNCTION_DECLARATION ::=
     function FUNCTION_IDENTIFIER
        (PARAMETER_IDENTIFIER : [in] String)
        return String;
```

The function can be an Ada function (default) or a shell script. To import a shell script, the pragma Import must be used:

```
PRAGMA ::=
   pragma Import
      ([Entity        =>] FUNCTION_IDENTIFIER,
       [Convention    =>] CONVENTION_LITERAL,
       [External_Name =>] STRING_LITERAL);

pragma Import (Best_Node, Shell, "best-node");
```

In this case, the PCS invokes the shell script with the partition name as a command line argument. The shell script is supposed to return the partition host-name (see Section 8.5.4.17 [Partition Attribute Host], page 83).

8.5.4.19 Partition Attribute Directory

Directory allows the user to specify in which directory the partition executable is stored. This can be useful in heterogeneous systems when the user wants to store executables for the same target in a given directory. Specifying the directory is also useful if the partition executable is not directly visible from the user environment. For instance, when a remote command like **rsh** is invoked, the executable directory has to be present in the user path. If the Directory attribute has been specified, the executable full name is used.

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Directory use STRING_LITERAL;
```

8.5.4.20 Partition Attribute Command_Line

The user may want to pass arguments on the command line of a partition. However, when a partition is launched automatically by the main partition, the partition command line includes only PolyORB arguments. To add arguments on the command line, the user can take advantage of the following attribute.

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Command_Line use STRING_LITERAL;
```

8.5.4.21 Partition Attribute Environment_Variables

The attribute Environment_Variables allows the user to specify a list of environment variables that should be passed from the main partition to slave partitions when using a generated (shell or Ada) launcher.

This attribute can be applied to all partitions by defining it for the predefined type **Partition**, or to a specific partition. Note that in the latter case, the list does not replace the default one but instead complements it (i.e. variables specified for **Partition** are passed in addition to the partition specific ones).

Use of this features requires that remote nodes provide the POSIX env(1) command.

```
STRING_LITERAL_LIST ::=
   STRING_LITERAL
   | STRING_LITERAL, STRING_LITERAL_LIST

REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Environment_Variables use (STRING_LITERAL_LIST);
```

8.5.4.22 Partition Attribute Termination

The Ada Reference Manual does not provide any specific rule to handle global termination of a distributed application (see Section 8.2.6 [Abortion and Termination], page 72).

In PolyORB/DSA, by default, a set of partitions terminates when each partition can terminate and when no message remains to be delivered. A distributed algorithm that checks for this global condition is activated periodically by the main boot server.

```
TERMINATION_LITERAL ::= Global_Termination |
                        Local_Termination  |
                        Deferred_Termination

REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Termination use TERMINATION_LITERAL;
```

- When a partition is configured with the global termination policy, it terminates as soon as the main boot server sends a signal to do so. The main boot server checks periodically whether the application can terminate. When all partitions are ready to terminate, the main boot server sends to each partition a termination request. The global termination policy is the default policy.
- The deferred termination policy is very similar to the global termination. The only difference is that when a partition with a deferred termination policy receives a termi-

nation request, it just ignores it. This policy allows a partition to run forever without preventing a set of partitions from terminating.

- When a partition is configured with the local termination policy, it terminates as soon as the classical Ada termination is detected by the partition. It means that this partition does not wait for the termination request of the main boot server.

8.5.4.23 Partition Attribute Reconnection

When no RCI package is configured on a partition, such a partition can be launched several times without any problem. When one or more RCI packages are configured on a partition, such a partition cannot be launched more than once. If this partition were to be launched repeatedly, it would not be possible to decide which partition instance should execute a remote procedure call.

When a partition crashes or is stopped, one may want to restart this partition and possibly restore its state - with Shared_Passive packages, for instance. In such a situation, the partition is already known to other partitions and possibly marked as a dead partition. Several policies can be selected:

```
RECONNECTION_LITERAL ::= Reject_On_Restart  |
                          Fail_Until_Restart |
                          Block_Until_Restart

REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Reconnection use RECONNECTION_LITERAL;
```

- When this partition is configured with the Reject_On_Restart reconnection policy, the dead partition is kept dead and any attempt to restart it fails. Any remote call to a subprogram located on this partition results in a Communication_Error exception. The Reject_On_Restart policy is the default policy.
- When this partition is configured with the Fail_Until_Restart reconnection policy, the dead partition can be restarted. Any remote call to a subprogram located on this partition results in an exception Communication_Error as long as this partition has not been restarted. As soon as the partition is restarted, remote calls to this partition are executed normally.
- When this partition is configured with the Block_Until_Restart reconnection policy, the dead partition partition can be restarted. Any remote call to a subprogram located on this partition is suspended until the partition is restarted. As soon as the partition is restarted, remote calls to this partition are executed normally. The suspended remote procedure calls to this partition are resumed.

8.5.4.24 Pragma Version

A library unit is consistent if the same version of its declaration is used throughout (see Section 8.2.5 [Consistency and Elaboration], page 71). It can be useful to deactivate these checks, especially when the user wants to be able to update a server without updating a client.

```
PRAGMA ::=
   pragma Version ([Check =>] BOOLEAN_LITERAL);
```

8.5.4.25 Partition Attribute Task_Pool

When multiple remote subprogram calls occur on the same partition, they are handled by several anonymous tasks. These tasks can be allocated dynamically or re-used from a pool of (preallocated) tasks. When a remote subprogram call is completed, the anonymous task can be deallocated or queued in a pool in order to be re-used for further remote subprogram calls. The number of tasks in the anonymous tasks pool can be configured by means of three independent parameters.

- The task pool minimum size indicates the number of anonymous tasks preallocated and always available in the PCS. Preallocating anonymous tasks can be useful in real-time systems to prevent task dynamic allocation.
- The task pool high size is a ceiling. When a remote subprogram call is completed, its anonymous task is deallocated if the number of tasks already in the pool is greater than the ceiling. If not, then the task is queued in the pool.
- The task pool maximum size indicates the maximum number of anonymous tasks in the PCS. In other words, it provides a way to limit the number of remote calls in the PCS. When a RPC request is received, if the number of active remote calls is greater than the task pool maximum size, then the request is kept pending until an anonymous task completes its own remote call and becomes available.

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Task_Pool use TASK_POOL_SIZE_ARRAY;

TASK_POOL_SIZE_ARRAY ::=
  (NATURAL_LITERAL,  --  Task Pool Minimum Size
   NATURAL_LITERAL,  --  Task Pool High Size
   NATURAL_LITERAL); --  Task Pool Maximum Size
```

In order to have only one active remote call at a time, the task pool configuration is declared as follows:

```
for Partition'Task_Pool use (0, 0, 1);
```

8.5.4.26 Partition Attribute ORB_Tasking_Policy

By default, the Thread_Pool ORB tasking policy is used for all partitions. This attribute allows selection of an alternate policy among those provided by PolyORB (see Section 5.2 [PolyORB ORB Tasking policies], page 21) for each partition.

```
ORB_TASKING_POLICY_LITERAL ::= Thread_Pool        |
                               Thread_Per_Session |
                               Thread_Per_Request

REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'ORB_Tasking_Policy use ORB_TASKING_POLICY_LITERAL;
```

Note: Section 8.5.4.25 [Partition Attribute Task_Pool], page 87 has no effect when another policy than Thread_Pool is activated.

8.5.4.27 A Complete Example

Almost every keyword and construct defined in the configuration language has been used in the following sample configuration file.

```
configuration MyConfig is

  Partition_1 : Partition := ();
  procedure Master_Procedure is in Partition_1;

  Partition_2, Partition_3 : Partition;

  for Partition_2'Host use "foo.bar.com";

  function Best_Node (Partition_Name : String) return String;
  pragma Import (Shell, Best_Node, "best-node");
  for Partition_3'Host use Best_Node;

  Partition_4 : Partition := (RCI_B5);

  for Partition_1'Directory use "/usr/you/test/bin";
  for Partition'Directory use "bin";

  procedure Another_Main;
  for Partition_3'Main use Another_Main;

  for Partition_3'Reconnection use Block_Until_Restart;
  for Partition_4'Command_Line use "-v";
  for Partition_4'Termination use Local_Termination;

  pragma Starter (Method => Ada);

  pragma Boot_Server
    (Protocol_Name => "tcp",
     Protocol_Data => "`hostname`:`unused-port`");

  pragma Version (False);

begin
   Partition_2 := (RCI_B2, RCI_B4, Normal);
   Partition_3 := (RCI_B3);
end MyConfig;
```

1. **Line 01** Typically, after having created the following configuration file the user types:

   ```
   po_gnatdist myconfig.cfg
   ```

   If the user wants to build only some partitions then he will list the partitions to build on the `po_gnatdist` command line as follows:

   ```
   po_gnatdist myconfig.cfg partition_2 partition_3
   ```

   The name of the file prefix must be the same as the name of the configuration unit, in this example `myconfig.cfg`. The file suffix must be `cfg`. For a given distributed application the user can have as many different configuration files as desired.

2. **Line 04** Partition 1 contains no RCI package. However, it will contain the main procedure of the distributed application, called *Master_Procedure* in this example. If the line *procedure Master_Procedure is in Partition_1;* was missing, Partition 1 would be completely empty. This is forbidden, because a partition has to contain at least one library unit.

   `po_gnatdist` produces an executable with the name of *Master_Procedure* which will start the various partitions on their host machines in the background. The main partition is launched in foreground. Note that by killing this main procedure the whole distributed application is terminated.
3. **Line 08** Specify the host on which to run partition 2.
4. **Line 12** Use the value returned by a program to figure out at execution time the name of the host on which partition 3 should execute. For instance, execute the shell script `best-node` which takes the partition name as parameter and returns a string giving the name of the machine on which partition_3 should be launched.
5. **Line 14** Partition 4 contains one RCI package RCI_B5 No host is specified for this partition. The startup script will ask for it interactively when it is executed.
6. **Line 16** Specify the directory in which the executable of partition partition_1 will be stored.
7. **Line 17** Specify the directory in which all the partition executables will be stored (except partition_1, see Section 8.5.4.2 [Pragmas and Representation Clauses], page 77). Default is the current directory.
8. **Line 20** Specify the partition main subprogram to use in a given partition.
9. **Line 22** Specify a reconnection policy in case of a crash of Partition_3. Any attempt to reconnect to Partition_3 when this partition is dead will be blocked until Partition_3 restarts. By default, any restart is rejected (Reject_On_Restart). Another policy is to raise Communication_Error on any reconnection attempt until Partition_3 has been restarted.
10. **Line 23** Specify additional arguments to pass on the command line when a given partition is launched.
11. **Line 24** Specify a termination mechanism for partition_4. The default is to compute a global distributed termination. When Local_Termination is specified a partition terminates as soon as local termination is detected (standard Ada termination).
12. **Line 26** Specify the kind of startup method the user wants. There are 3 possibilities: Shell, Ada and None. Specifying *Shell* builds a shell script. All the partitions will be launched from a shell script. If *Ada* is chosen, then the main Ada procedure itself is used to launch the various partitions. If method *None* is chosen, then no launch method is used and the user must start each partition manually.

    If no starter is given, then an Ada starter will be used.

    In this example, Partition_2, Partitions_3 and Partition_4 will be started from Partition_1 (ie from the Ada procedure Master_Procedure).
13. **Line 30** Specify the use of a particular boot server.
14. **Line 32** It is a bounded error to elaborate a partition of a distributed program that contains a compilation unit that depends on a different version of the declaration of an RCI library unit than the one included in the partition to which the RCI library

unit was assigned. When the pragma Version is set to False, no consistency check is performed.

15. **Line 34** The configuration body is optional. The user may have fully described his configuration in the declaration part.
16. **Line 35** Partition 2 contains two RCI packages RCI_B2 and RCI_B4 and a normal package. A normal package is not categorized.
17. **Line 36** Partition 3 contains one RCI package RCI_B3

8.5.5 Partition Runtime Parameters

You can adjust some parameters of your DSA applications using the PolyORB configuration file, '`polyorb.conf`'. The parameters relevant to the Ada Distributed Systems Annex are specified in the `[dsa]` section.

See Section 4.2 [Run-time configuration], page 14 for complete documentation of PolyORB's runtime configuration facilities.

`name_service = [IOR/corbaloc]`
: You can set this parameter instead of the environment variable `POLYORB_DSA_NAME_SERVICE`. Though if you use a Starter, ensure that this parameter is set for all the partitions, as this is not done automatically as for the `POLYORB_DSA_NAME_SERVICE` environment variable.

`max_failed_requests = [integer]`
: Each partition will attempt a given number of requests to the name server before failing. This allows some time for every partition to register in the name server.

`delay_between_failed_requests = [duration in milliseconds]`
: As above, only this specifies the delay between requests.

`termination_initiator = [true/false]`
: Is this partition a termination initiator.

`termination_policy = [global_termination/deferred_termination/local_termination]`
: The termination policy for this partition.

`tm_time_between_waves = [duration in milliseconds]`
: The delay between termination waves.

`tm_time_before_start = [duration in milliseconds]`
: The delay before the termination manager starts sending waves.

`detach = [true/false]`
: If true, the partition will be detached.

`rsh_options = [string]`
: Options passed to the rsh command when using the module polyorb.dsa_p-remote_launch

`rsh_command = [string]`
: Which command should the module polyorb.dsa_p-remote_launch use to spawn remote programs.

8.5.6 Gnatdist Internals

Here is what goes on in `po_gnatdist` when building a distributed application:

- Each compilation unit in the program is compiled into an object module (as for non distributed applications). This is achieved by calling gnatmake on the sources of the various partitions.
- Stubs and skeletons are compiled into object modules (these are pieces of code that allow a partition running on machine A to communicate with a partition running on machine B). Several timestamp checks are performed to avoid useless code recompilation and stub generation.
- `po_gnatdist` performs a number of consistency checks. For instance it checks that all packages marked as remote call interface (RCI) and shared passive (SP) are mapped onto partitions. It also checks that a RCI or SP package is mapped onto only one partition.
- Finally, the executables for each partition in the program are created. The code to launch partitions is embedded in the main partition except if another option has been specified (see Section 8.5.4.7 [Pragma Starter], page 80). In this case, a shell script (or nothing) is generated to start the partitions on the appropriate machines. This is specially useful when one wants to write client / server applications where the number of instances of the partition is unknown.

All Gnatdist intermediate files (object files, etc) are stored under a common directory named `"dsa"`. The user may remove this whole directory and its content when he does not intend to rebuild his distributed applications.

8.5.7 PolyORB PCS Internals

This section provides notes on the PolyORB implementation of the DSA PCS. Some of these features are not configurable by the user.

8.5.7.1 Application Startup

A name server needs to be started prior to starting any application partition. Once the name server is started, its location must be passed to all partitions as the `name_service` runtime parameter in the `[dsa]` section of the configuration. When using an Ada starter, it is sufficient to pass the name server location to the starter, and it will be propagated automatically to all partitions.

Upon elaboration, each partition registers its RCI packages with the name server. Once this is done, remote calls to RCI subprograms can proceed. Partitions cache the replies from the name server so that during the course of normal execution, inter-partition calls only involve the caller and callee partitions (not the name server).

RCI units can then act as "clearinghouses" for other partitions to exchange RACWs and set up dynamic communication paths.

8.5.7.2 Heterogeneous System

The GNAT environment provides default stream attributes, except for non-remote access types (see Section 8.1.5.3 [Transmitting Dynamic Structure], page 61 and Section 8.2.1 [Marshalling and Unmarshalling Operations], page 69). The implementation of the default attributes of predefined types can be found in *System.Stream_Attributes* (s-stratt.adb).

The PolyORB PCS provides alternative data representations by default to ensure portability of the data stream across partitions executing on heterogeneous architectures. Users may override these representation aspects by configuring the protocol personality of their choice.

8.5.7.3 Allocating Partition Ids

The Partition_ID is allocated dynamically, at run-time. Each partition connects to a Partition ID Server which is located on the boot server and asks for a free Partition_ID. The advantage of this approach is that it supports easily client / server solution (client partitions may be duplicated, they will obtain different Partition Ids). There is no need to recompile or relink all the partitions when a new partition is added to the system. The Partition_ID is not tied in any way to a specific protocol or to a specific location.

8.5.7.4 Executing Concurrent Remote Calls

When multiple remote subprogram calls occur on the same partition, they are handled by several anonymous tasks. The number of tasks in the anonymous tasks pool can be configured by three figures (see Section 8.5.4.25 [Partition Attribute Task_Pool], page 87). Therefore, the user may have to synchronize global data in the Remote_Call_Interface or Remote_Types unit to preserve concurrent access on data. If the user want to suppress the multiple requests features, he can force the configuration of the anonymous tasks pool to (0 `|` 1, 0 `|` 1, 1). That means that there will be at most one anonymous task running at a time.

8.5.7.5 Priority Inheritance

It is compiler-dependent whether the caller priority is preserved during a remote procedure call. In fact, it can be unsafe to rely on priorities, because two partitions may have different priority ranges and policies. Nevertheless, PolyORB preserves the caller priority. This priority is marshaled and unmarshaled during the remote procedure call and the priority of the anonymous task on the server is set to the caller priority.

This default policy can be modified by using pragma Priority Section 8.5.4.15 [Pragma Priority], page 83 and partition attribute Priority Section 8.5.4.16 [Partition Attribute Priority], page 83.

8.5.7.6 Remote Call Abortion

When a remote procedure call is aborted, PolyORB will abort the calling task on the caller side. It will also try to abort the remote anonymous task performing the remote call, unless runtime parameter `abortable_rpcs` in section `[tasking]` is set False on the server.

8.6 Running a DSA application

By default `po_gnatdist` will use the Ada starter. So if you have not specified `pragma Starter (None);` in the `po_gnatdist` configuration file, you should have a starter in your build directory, named after the main procedure defined in the configuration file. In this case you just have to run this program.

If you don't want to use the Starter and have specified `pragma Starter (None);` in your configuration file, then you should have, in your Partition'Directory, one binary for each of your partitions. You'll have to start each of these programs manually.

In both cases you must specify a name server for your application. You can use for example the one included in PolyORB: 'po_cos_naming'. When running this name server it will output its IOR URI named POLYORB_CORBA_NAME_SERVICE.

Just ensure that you set the global environment variable POLYORB_DSA_NAME_SERVICE to an IOR URI referencing the running name server. When using the 'po_cos_naming' name server just set POLYORB_DSA_NAME_SERVICE environment variable to the first value output for POLYORB_DSA_NAME_SERVICE before launching each DSA partition.

Here is a small trace output that demonstrates the setup

```
polyorb/examples/dsa/echo% ../../../tools/po_cos_naming/po_cos_naming&
polyorb/examples/dsa/echo% POLYORB_CORBA_NAME_SERVICE=''....''

polyorb/examples/dsa/echo% export POLYORB_DSA_NAME_SERVICE=''...''
polyorb/examples/dsa/echo% ./client
The client has started!
Thus spake my server upon me:Hi!
```

9 MOMA

9.1 What you should know before Reading this section

This section assumes that the reader is familiar with the JMS specifications described in *[SUN99]*. MOMA is a thick adaptation of the JMS specification to the Ada programming language, preserving most of the JMS concepts.

9.2 Installing MOMA application personality

Ensure PolyORB has been configured and then compiled with the MOMA application personality. See Chapter 4 [Building an application with PolyORB], page 13 for more details on how to check installed personalities.

To build the MOMA application personality, see Chapter 2 [Installation], page 7.

9.3 Package hierarchy

Packages installed in '`$INSTALL_DIR/include/polyorb/moma`' hold the MOMA API. MOMA is built around two distinct sets of packages:

1. '`MOMA.*`' hold the public MOMA library, all the constructs the user may use.
2. '`POLYORB.MOMA_P.*`' hold the private MOMA library, these packages shall not be used when building your application.

10 Ada Web Server (AWS)

The documentation of this personality will appear in a future revision of PolyORB.

11 GIOP

11.1 Installing GIOP protocol personality

Ensure PolyORB has been configured and then compiled with the GIOP protocol personality. See Chapter 4 [Building an application with PolyORB], page 13 for more details on how to check installed personalities.

To enable configuration of the GIOP protocol personality, see Chapter 2 [Installation], page 7.

11.2 GIOP Instances

GIOP is a generic protocol that can be instantiated for multiple transport stacks. PolyORB provides three different instances.

11.2.1 IIOP

Internet Inter-ORB Protocol (IIOP) is the default protocol defined by the CORBA specifications. It is a TCP/IP, IPv4, based protocol that supports the full semantics of CORBA requests.

11.2.2 SSLIOP

The SSLIOP protocol provides transport layer security for transmitted requests. Its provides encryption of GIOP requests.

To build the SSLIOP, it is required to activate SSL-related features when building PolyORB. See '`--with-openssl`' in Chapter 2 [Installation], page 7 for more details.

Enabling security is completely transparent to a preexisting application, it is also possible to phase in secure communications by allowing incoming requests which are unsecured.

11.2.3 DIOP

Datagram Inter-ORB Protocol (DIOP) is a specialization of GIOP for the UDP/IP protocol stack. It supports only asynchronous (`oneway`) requests.

This protocol is specific to PolyORB. DIOP 1.0 is a mapping of GIOP on top of UDP/IP. DIOP 1.0 uses GIOP 1.2 message format.

11.2.4 MIOP

Unreliable Multicast Inter-ORB Protocol (MIOP) *[OMG02b]* is a specialization of GIOP for IP/multicast protocol stack. It supports only asynchronous (`oneway`) requests.

11.3 Configuring the GIOP personality

The GIOP personality is configured using a configuration file. See Section 4.2.1 [Using a configuration file], page 14 for more details.

Here is a summary of available parameters for each instance of GIOP.

11.3.1 Common configuration parameters

This section details configuration parameters common to all GIOP instances.

```
###############################################################################
# GIOP parameters
#

[giop]

###############################################################
# Native code sets
#
# Available char data code sets:
#   16#00010001#   ISO 8859-1:1987; Latin Alphabet No. 1
#   16#05010001#   X/Open UTF-8; UCS Transformation Format 8 (UTF-8)
#
# Available wchar data code sets:
#   16#00010100#   ISO/IEC 10646-1:1993; UCS-2, Level 1
#   16#00010109#   ISO/IEC 10646-1:1993;
#                               UTF-16, UCS Transformation Format 16-bit form
#
#giop.native_char_code_set=16#00010001#
#giop.native_wchar_code_set=16#00010100#
#
# The following parameters force the inclusion of fallback code sets
# as supported conversion code sets. This is required to enable
# interoperability with ORBs whose code sets negotiation support is
# broken. See PolyORB's Users Guide for additional information.
#
#giop.add_char_fallback_code_set=false
#giop.add_wchar_fallback_code_set=false
```

11.3.2 IIOP Configuration Parameters

```
###############################################################################
# IIOP parameters
#

[iiop]

###############################################################
# IIOP Global Settings

# Preference level for IIOP
#polyorb.binding_data.iiop.preference=0

# IIOP's default address
#polyorb.protocols.iiop.default_addr=127.0.0.1

# IIOP's default port
#polyorb.protocols.iiop.default_port=2809

# IIOP's alternate addresses
#polyorb.protocols.iiop.alternate_listen_addresses=127.0.0.1:2810 127.0.0.1:2820

# Default GIOP/IIOP Version
#polyorb.protocols.iiop.giop.default_version.major=1
#polyorb.protocols.iiop.giop.default_version.minor=2
```

```
###############################################################
# IIOP 1.2 specific parameters

# Set to True to enable IIOP 1.2
#polyorb.protocols.iiop.giop.1.2.enable=true

# Set to True to send a locate message prior to the request
#polyorb.protocols.iiop.giop.1.2.locate_then_request=true

# Maximum message size before fragmenting request
#polyorb.protocols.iiop.giop.1.2.max_message_size=1000

###############################################################
# IIOP 1.1 specific parameters

# Set to True to enable IIOP 1.1
#polyorb.protocols.iiop.giop.1.1.enable=true

# Set to True to send a locate message prior to the request
#polyorb.protocols.iiop.giop.1.1.locate_then_request=true

# Maximum message size before fragmenting request
#polyorb.protocols.iiop.giop.1.1.max_message_size=1000

###############################################################
# IIOP 1.0 specific parameters

# Set to True to enable IIOP 1.0
#polyorb.protocols.iiop.giop.1.0.enable=true

# Set to True to send a locate message prior to the request
#polyorb.protocols.iiop.giop.1.0.locate_then_request=true
```

11.3.3 SSLIOP Configuration Parameters

11.3.3.1 Ciphers name

PolyORB's SSLIOP uses the OpenSSL library to support all ciphers recommended by CORBA 3.0.3. The OpenSSL library uses specific names for ciphers. The table below contains CORBA-recommended cipher names and their OpenSSL equivalents:

CORBA recommended ciphers	OpenSSL equivalent
TLS_RSA_WITH_RC4_128_MD5	RC4-MD5
SSL_RSA_WITH_RC4_128_MD5	RC4-MD5
TLS_DHE_DSS_WITH_DES_CBC_SHA	EDH-DSS-CBC-SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA	EDH-DSS-CBC-SHA
TLS_RSA_EXPORT_WITH_RC4_40_MD5	EXP-RC4-MD5
SSL_RSA_EXPORT_WITH_RC4_40_MD5	EXP-RC4-MD5
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	EXP-EDH-DSS-DES-CBC-SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	EXP-EDH-DSS-DES-CBC-SHA

11.3.3.2 SSLIOP Parameters

```
###############################################################################
# SSLIOP parameters
#
```

```
[ssliop]

###############################################################
# SSLIOP Global Settings

# SSLIOP's default port
#polyorb.protocols.ssliop.default_port=2810
# If no SSLIOP default address is provide, PolyORB reuses IIOP's
# address

# Private Key file name
#polyorb.protocols.ssliop.privatekeyfile=privkey.pem

# Certificate file name
#polyorb.protocols.ssliop.certificatefile=cert.pem

# Trusted CA certificates file
#polyorb.protocols.ssliop.cafile=cacert.pem

# Trusted CA certificates path
#polyorb.protocols.ssliop.capath=demoCA/certs

# Disable unprotected invocations
#polyorb.protocols.ssliop.disable_unprotected_invocations=true

###############################################################
# Peer certificate verification mode

# Verify peer certificate
#polyorb.protocols.ssliop.verify=false

# Fail if client did not return certificate. (server side option)
#polyorb.protocols.ssliop.verify_fail_if_no_peer_cert=false

# Request client certificate only once. (server side option)
#polyorb.protocols.ssliop.verify_client_once=false
```

11.3.4 DIOP Configuration Parameters

```
###############################################################
# DIOP Global Settings

# Preference level for DIOP
#polyorb.binding_data.diop.preference=0

# DIOP's default address
#polyorb.protocols.diop.default_addr=127.0.0.1

# DIOP's default port
#polyorb.protocols.diop.default_port=12345

# Default GIOP/DIOP Version
#polyorb.protocols.diop.giop.default_version.major=1
#polyorb.protocols.diop.giop.default_version.minor=2

###############################################################
# DIOP 1.2 specific parameters
```

```
# Set to True to enable DIOP 1.2
#polyorb.protocols.diop.giop.1.2.enable=true

# Maximum message size
#polyorb.protocols.diop.giop.1.2.max_message_size=1000

###############################################################
# DIOP 1.1 specific parameters

# Set to True to enable DIOP 1.1
#polyorb.protocols.diop.giop.1.1.enable=true

# Maximum message size
#polyorb.protocols.diop.giop.1.1.max_message_size=1000

###############################################################
# DIOP 1.0 specific parameters

# Set to True to enable DIOP 1.0
#polyorb.protocols.diop.giop.1.0.enable=true
```

11.3.5 MIOP Configuration Parameters

```
###############################################################################
# MIOP parameters
#

[miop]

###############################################################
# MIOP Global Settings

# Preference level for MIOP
#polyorb.binding_data.uipmc.preference=0

# Maximum message size
#polyorb.miop.max_message_size=6000

# Time To Leave parameter
#polyorb.miop.ttl=15

# Multicast address to use
# These two parameters must be set explicitly, no default value is provided.
# If either parameter is unset, the MIOP access point is disabled.
#polyorb.miop.multicast_addr=<group-ip-address>
#polyorb.miop.multicast_port=<port-number>

# Set to True to enable MIOP
#polyorb.protocols.miop.giop.1.2.enable=false

# Maximum message size
#polyorb.protocols.miop.giop.1.2.max_message_size=1000
```

11.4 Code sets

This sections details the various steps required to add support for new character code sets to PolyORB's GIOP personality. Please refer to the CORBA specifications (*[OMG04]*), par. 13.10 for more details on this topic.

11.4.1 Supported code sets

PolyORB supports the following list of code sets:

1. Available char data code sets:
   1. 16#00010001# ISO 8859-1:1987; Latin Alphabet No. 1
   2. 16#05010001# X/Open UTF-8; UCS Transformation Format 8 (UTF-8)
2. Available wchar data code sets:
   1. 16#00010100# ISO/IEC 10646-1:1993; UCS-2, Level 1
   2. 16#00010109# ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format 16-bit form

11.4.2 Incompatibility in code set support

Some ORBs report incompatiblity in code sets because fallback converters are not explicitly present in the reference. To work around this issue, you may use the following parameters:

```
[giop]
giop.add_char_fallback_code_set=true
giop.add_wchar_fallback_code_set=true
```

11.4.3 Adding support for new code sets

PolyORB allows users to extend the set of supported native character code sets. Adding support for new character code set consists of the following steps:

1. Developing sets of Converters - special objects that do marshalling/unmarshalling operations of character data. At least two Converters are required: for direct marshalling character data in native code set and for marshalling/unmarshalling character data in fallback character code set (UTF-8 for char data and UTF-16 for wchar data). Additional Converters may be developed for marshalling character data in conversion code set.
2. Developing converter factory subprogram for each Converter.
3. Registering native code set, its native and fallback converters and optional conversion char sets and its converters.

11.4.4 Character data Converter

Character data converters do direct marshalling/unmarshalling of character data (char or wchar - depending on `Converter`) into/from PolyORB's buffer. This allows to minimize the speed penalty on character data marshalling.

Character data Converters for char data have the following API (from '`PolyORB.GIOP_P.Code_Sets.Converters`' package:

```
type Converter is abstract tagged private;

procedure Marshall
  (C      : Converter;
   Buffer : access Buffers.Buffer_Type;
   Data   : Types.Char;
   Error  : in out Errors.Error_Container)
   is abstract;

procedure Marshall
```

```
  (C      : Converter;
   Buffer : access Buffers.Buffer_Type;
   Data   : Types.String;
   Error  : in out Errors.Error_Container)
   is abstract;

procedure Unmarshall
  (C      : Converter;
   Buffer : access Buffers.Buffer_Type;
   Data   :    out Types.Char;
   Error  : in out Errors.Error_Container)
   is abstract;

procedure Unmarshall
  (C      : Converter;
   Buffer : access Buffers.Buffer_Type;
   Data   :    out Types.String;
   Error  : in out Errors.Error_Container)
   is abstract;
```

The Marshall subprograms do marshalling of one character or string of characters into the buffer. The Unmarshall subprograms do unmarshalling of one character or string of characters from the buffer.

Note: Depending on the item size of the data (char/wchar) and GIOP version, marshalling/unmarshalling algorithms may vary. In some situations marshalling of string is not equivalent to marshalling its length and marshalling one by one each character. Please refere to GIOP specifications for more details.

If marshalling/unmarshalling fails, subprograms must set the Error parameter to the corresponding error, usually `Data_Conversion_E`.

Note: We recommend to always use the Data_Conversion_E error code with Minor status 1.

All `Converters` (native, fallback and conversion) have similar APIs. Wchar data converters differ only in parameter type.

11.4.5 Converters factories

To create new converters, PolyORB uses special factory subprograms with the following profile:

```
function Factory return Converter_Access;
```

or

```
function Factory return Wide_Converter_Access;
```

This function must allocate a new `Converter` and initialize its state.

11.4.6 Registering new code sets

Registering new native character data code sets begins from registering new native character data code sets and its native and fallback `Converters`. This is done using `Register_Native_Code_Set`:

```
procedure Register_Native_Code_Set
  (Code_Set : Code_Set_Id;
   Native   : Converter_Factory;
   Fallback : Converter_Factory);
```

or

```
procedure Register_Native_Code_Set
  (Code_Set : Code_Set_Id;
   Native   : Wide_Converter_Factory;
   Fallback : Wide_Converter_Factory);
```

If you have additional conversion code sets Converters you may register it by calling Register_Conversion_Code_Set subprogram:

```
procedure Register_Conversion_Code_Set
  (Native     : Code_Set_Id;
   Conversion : Code_Set_Id;
   Factory    : Converter_Factory);
```

or

```
procedure Register_Conversion_Code_Set
  (Native     : Code_Set_Id;
   Conversion : Code_Set_Id;
   Factory    : Wide_Converter_Factory);
```

Note: because of incompatibility in the support of code sets negotiation in some ORB's it is recommend to recognize two boolean PolyORB's parameters:

```
[giop]
giop.add_char_fallback_code_set=false
giop.add_wchar_fallback_code_set=false
```

and also register a fallback Converter as conversion Converter if the corresponding parameter is set to True.

Finally, define your preferred native character data code sets by parameters (only integer code sets codes now supported):

```
[giop]
giop.native_char_code_set=16#00010001#
giop.native_wchar_code_set=16#00010100#
```

12 SOAP

12.1 Installing SOAP protocol personality

Ensure PolyORB has been configured and then compiled with the SOAP protocol personality. See Chapter 4 [Building an application with PolyORB], page 13 for more details on how to check installed personalities.

To enable configuration of the SOAP application personality, see Chapter 2 [Installation], page 7.

12.2 Configuring the SOAP personality

The SOAP personality is configured using a configuration file. See Section 4.2.1 [Using a configuration file], page 14 for more details.

Here is a summary of available parameters for each instance of SOAP.

```
###############################################################################
# SOAP parameters
#

[soap]

###############################################################
# SOAP Global Settings

# Preference level for SOAP
#polyorb.binding_data.soap.preference=0

# SOAP's default address
#polyorb.protocols.soap.default_addr=127.0.0.1

# SOAP's default port
#polyorb.protocols.soap.default_port=8080
```

13 Tools

13.1 po_catref

`po_catref` is a utility for viewing the components of a stringified reference (CORBA IOR, corbaloc or URI). The reference's components include references to access an object through multiple protocols (e.g. CORBA IIOP, SOAP) and configuration parameters associated with a reference (e.g. GIOP Service Contexts).

```
Usage:
          po_catref <stringified reference>
```

Note: `po_catref` *can only process protocols PolyORB has been configured with.*

13.2 po_dumpir

`po_dumpir` is a utility for viewing the content of an instance of the CORBA Interface Repository.

```
Usage:
          po_dumpir <stringified reference>
```

Note: `po_dumpir` *will be compiled and installed only if the CORBA personality and the* '`ir`' *service is compiled. Please see* *Chapter 4 [Building an application with PolyORB], page 13* *for more details on how to set up PolyORB.*

13.3 po_names

`po_names` is a stand-alone name server. It has an interface similar to CORBA COS Naming, without dragging in any dependencies on CORBA mechanisms. This name server is to be used when the CORBA application personality is not required, e.g. with the DSA or MOMA application personalities.

Appendix A Performance considerations

This section discusses performance when using PolyORB. Many elements can be configured, See Chapter 4 [Building an application with PolyORB], page 13. By carefully selecting them, you can increase the throughput of your application.

We review some parameters that can impact performance.

- **Build options**:
  - For production use, you should not build PolyORB with debug activated.
- **Tasking policies**:
  - You should carefully select the tasking policy to reduce dynamic ressource allocation (tasks, entry points, etc.). See Chapter 5 [Tasking model in PolyORB], page 19.
- **Transport parameters**:
  - Setting `tcp.nodelay` to false will disable Nagle buffering.
- **GIOP parameters**:
  - Setting `polyorb.protocols.iiop.giop.1.X.locate_then_request`, where `X` is the GIOP version in use, to false will disable `Locate_Message`, reducing the number of requests exchanged,
  - Increasing `polyorb.protocols.iiop.giop.1.X.max_message_size`, where `X` is the GIOP version in use, will reduce GIOP fragmentation, reducing middleware processing.

Appendix B Conformance to standards

B.1 CORBA standards conformance

The OMG defines a CORBA-compliant ORB as an implementation of the CORBA specifications that supports CORBA Core and one mapping of CORBA's IDL.

Here is a summary of PolyORB's conformance issues with the latest CORBA specifications (revision 3.0, formal/02-06-01).

B.1.1 CORBA IDL-to-Ada mapping

PolyORB supports the IDL-to-Ada specification *[OMG01]*, with the following limitations in both the CORBA API and the IDL-to-Ada compiler `idlac`:

- no support for abstract interfaces, object-by-value, context data;
- no support for CORBA Components;
- implemented API may present some divergences with current mapping.

Note: generated code is constrained by the limitations of the Ada compiler used. Please refer to its documentation for more information.

Conforming to documentation requirements from section 4.11 of the IDL-to-Ada specification *[OMG01]*, note that PolyORB's implementation of CORBA is *tasking-safe*. The use of the CORBA personality on typical GNAT runtimes is *task-blocking*, unless specified in platform notes.

B.1.2 CORBA Core

This set encompasses chapters 1-11. Chapters 3 to 11 are normative.

- Chapter 3 describes OMG IDL syntax and semantics. See Section B.1.1 [CORBA IDL-to-Ada mapping], page 113 for a description of non-implemented features;
- Chapter 4 describes the ORB Interface.

  PolyORB partially supports this chapter.
- Chapter 5 describes Value Type Semantics.

  PolyORB does not support this chapter.
- Chapter 6 describes Abstract Interface Semantics.

  PolyORB does not support this chapter.
- Chapter 7 describes Dynamic Invocation Interface (DII)

  PolyORB supports only the following methods: `Create_Request`, `Invoke` and `Delete`.
- Chapter 8 describes Dynamic Skeleton Interface (DSI)

  PolyORB partially supports this chapter: this interface is fully implemented except for context data.
- Chapter 9 describes Dynamic Management of Any Values

  PolyORB partially supports this chapter: this interface is fully implemented except for object references and value types.

- Chapter 10 describes The Interface Repository

  PolyORB supports this chapter, except for the `ExtValueDef` interface, and all CORBA CCM related interfaces.
- Chapter 11 describes The Portable Object Adapter

  PolyORB supports this chapter with the following limitations:
  - the `USE_SERVANT_MANAGER` policy is partially supported: the `ServantLocator` object is not implemented;
  - support for `SINGLE_THREAD` policy is incomplete, reentrant calls may not work;
  - `Wait_For_Completion` and `Etherealize_Objects` are not taken into account in `PortableServer.POAManager`;
  - the `PortableServer.POAManagerFactory` API is not implemented.

B.1.3 CORBA Interoperability

This set encompasses chapters 12-16.

- See Section B.4 [CORBA-GIOP standards conformance], page 115 for more information on this point.

B.1.4 CORBA Interworking

This set encompasses chapters 17-21.

- Chapters 17 to 20 describe interoperability with Microsoft's COM/DCOM.

  PolyORB provides no support for these chapters.
- Chapter 21 describes `PortableInterceptor`.

  PolyORB provides partial support for this chapter.

B.1.5 CORBA Quality Of Service

This set encompasses chapters 22-24.

- Chapter 22 describes CORBA Messaging
- Chapter 23 describes Fault Tolerant CORBA
- Chapter 24 describes Secure Interoperability.

PolyORB provides no support for these chapters.

B.1.6 CORBA COS Services

COS Services are specifications of high level services that are optional extensions to the CORBA specification. They provide helper packages to build distributed applications. PolyORB implements the following COS Services:

- COS Event and TypedEvent;
- COS Naming;
- COS Notification;
- COS Time;

B.1.7 CORBA Specialized services

PolyORB supports the following specialized services:

- Unreliable Multicast (MIOP), proposed 1.0 specification *[OMG02b]*.
- RT-CORBA extensions, see Chapter 7 [RT-CORBA], page 49 for more information on this point.
- CORBA security extensions, see *[OMG]* for more information on this point.

B.2 RT-CORBA standards conformance

RT-CORBA specifications rely on the CORBA application personality; the same issues and implementation notes apply.

In addition, here is a list of issues with the implementation of RT-CORBA static *[OMG02a]* and dynamic scheduling *[OMG03]* specifications.

- RT-CORBA static and dynamic scheduling (Chapter 2)

  Chapter 2 is common to these two specifications. It describes key mechanisms of RT-CORBA that are common to both specifications.

  PolyORB partially implements this chapter from section 2.1 up to section 2.10. PolyORB does not provide support for all connection-related policies.

  See implementation notes in the different package specifications for more details.
- RT-CORBA static scheduling (Chapter 3)

  PolyORB supports this chapter.
- RT-CORBA dynamic scheduling (Chapter 3)

  PolyORB does not support this chapter.

B.3 CSIv2 standards conformance

PolyORB supports IIOP/SSL.

B.4 CORBA/GIOP standards conformance

GIOP supports part of the CORBA Interoperability specification, from chapters 12 to 16 of CORBA specifications.

Chapter 12 defines general concepts about ORB interoperability. It defines an *interoperbility-compliant ORB* as an ORB that supports:

- API that supports the construction of request-level inter-ORB bridges, Dynamic Invocation Interface, Dynamic Skeleton Interface and the object identity operations described in the Interface Repository. See Section B.1 [CORBA standards conformance], page 113 for more details.
- IIOP protocol as defined in chapter 15.

Support for other components is optional.

- Chapter 13 describes the ORB Interoperability Architecture.

  PolyORB fully supports this chapter.
- Chapter 14 describes how to build Inter-ORB Bridges.

  PolyORB fully supports this chapter.

- Chapter 15 describes the General Inter-ORB Protocol (GIOP).

  PolyORB supports GIOP version 1.0 to 1.2, the CDR representation scheme. Support for IOR and `corbaloc` addressing mechanisms is supported in the CORBA personality, see Chapter 6 [CORBA], page 25 for more details.

  PolyORB does not support the optional IIOP IOR Profile Components, Bi-directional GIOP. PolyORB also does not support fragmentation in GIOP 1.1.
- Chapter 16 describes the DCE ESIOP protocol.

  PolyORB does not support this optional chapter.

B.5 SOAP standards conformance

The documentation of the SOAP standards conformance of PolyORB will appear in a future revision of PolyORB.

Appendix C References

1. [DB98] B. Dobbing and A. Burns. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of SigAda'98*, Washington, DC, USA, November 1998.
2. [gla06] *GLADE User's Guide r 1.54*, February 2006.
3. [ISO95] ISO. *Information Technology – Programming Languages – Ada*. ISO, February 1995. ISO/IEC/ANSI 8652:1995.
4. [ISO06] ISO. *Information Technology – Programming Languages – Ada*. ISO, November 2006. ISO/IEC 8652:1995 with Technical Corrigendum 1 and Amendment 1 ("Ada 2005").
5. [Obr03] P. Obry. Ada Web Server (AWS) 1.3, 2003.
6. [OMG] OMG. *Common Secure Interoperability (CSIv2)*. OMG.
7. [OMG01] OMG. *Ada Language Mapping Specification, v1.2*. OMG, October 2001. OMG Technical Document formal/2001-10-42.
8. [OMG02a] OMG. *Real-Time CORBA Specification, static scheduling, v1.1*. OMG, April 2002. OMG Technical Document formal/2002-08-02.
9. [OMG02b] OMG. *unreliable Multicast InterORB Protocol specification*. OMG, 2002. OMG Technical Document ptc/03-01-11.
10. [OMG03] OMG. *Real-Time CORBA Specification, dynamic scheduling, v2.0*. OMG, April 2003. OMG Technical Document formal/2003-11-01.
11. [OMG04] OMG. *The Common Object Request Broker: Architecture and Specification, revision 3.0.3*. OMG, March 2004. OMG Technical Document formal/2004-03-12.
12. [SUN99] SUN. Java Message Service, 1999.
13. [W3C00] W3C. *Extensible Markup Language (XML) 1.0*, October 2000. W3C recommandation.
14. [W3C03] W3C. *Simple Object Access Protocol (SOAP) 1.2: primer*, june 2003. W3C recommandation.

Appendix D GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the

Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Heading 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute

the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index